SCOrWare Project

Conception and Development Tools Specifications

Version 2.0

Funded by the French National Research Agency

Colophon

Date	March 20 2009
Deliverable type	Specification
Version	2.0
Status	Revision
Work Package	2
Access Permissions	Public

Editor

Obeo	Stéphane Drapeau

Authors

EBM WebSourcing	Vincent Zurczak
INRIA	Damien Fournier Philippe Merle
IRIT	Aurélie Hurault
INT	Djamel Belaid Samir Tata
Obeo	Stéphane Drapeau
Open Wide	Marc Dutoo

Table of Contents

List of Figures

1 Introduction

The SCOrWare project objectives are to develop:

- A runtime platform for deploying, executing, and managing SCA based applications.
- A set of development tools for modeling, designing, and implementing SCA based applications.
- A set of demonstrators.

This document specifies the set of tools used to design, develop, test and deploy elements to build a distributed architecture compound of components, services, and business services. These tools are based on standards like MDA (Model Driven Architecture) [1], DSL (Domain Specific Language) [2], SCA (Service Component Architecture) and are build on top of Eclipse. Some of these tools complete currently available tools proposed in the SOA Tools Platform (STP) project [3], Java Workflow Tools (JWT) project and others are specific to the SCOrWare project. Section 1.1 lists existing STP tools, and Section 1.2 gives an overview of the contributions of the developed SCOrWare tools, and a summary of next chapters.

1.1 Eclipse SOA Tools Platform Project

STP sub projects

The goal of STP is to provide an integrated developer tooling platform for infrastructures based on Service Oriented Architecture (SOA). STP is composed of 3 retired sub-projects and 6 active sub-projects:

1. **Retired - Core Frameworks** (CF) defines EMF models that conform to the SCA specification for service assembly as well as Java language components for SCA syntax support.
2. **Retired - SOA System** (SOAS) provides tools and frameworks for assembling, building, packaging and deploying services to runtime containers. In addition support is provided for the definition and association of policy to services prior to deployment.
3. **Retired - Service Creation** (SC) handles the management of the relationship between the SOA model tooling provided by STP and the actual implementation tooling(s). The ultimate goal is to support the development of SOA network via tools fully integrated for top-down and bottom-up approaches as well as a mix of both in an agile way.
4. **BPEL 2 Java** (B2J) provides tools to translate BPEL into executable Java classes. It also defines a standard framework to which these executable Java classes can be deployed.
5. **BPMN** (BPMN) provides an editor and a set of tools to model business processes diagrams using the BPMN notation.
6. **SCA Tools project** provides a set of tools to construct top-down and bottom-up SCA composites. A large part of this project is implemented in the scope of the SCOrWare project.
7. **Enterprise Integration Patterns Editor** provides an editor for Enterprise integration patterns that has attached a generation framework that can produce standard JBI 1.0 components for deployment.
8. **SOA Policy Editor** allows editing WS-Policy standard documents and assertions. A first contribution is a form-based editor that integrates representations of policy instances with documentation. A second contribution is an extension of the WTP WSDL editor and usuals a similar visual metaphor for interactions.
9. **SOA intermediate model** is a meta model that replaces the SCA meta model used in the Core Framework sub-project. This contribution is described in the next paragraph about reconsideration of the meta model to use.

Reconsideration of the meta model to use

At the core of STP was the SCA meta model (the core framework sub-project) which was the basis for all tools. However, the SCA meta model code has not received much attention since the initial contribution, it is out of date and does not appear to be an appropriate match to the sub-project requirements for modelling. Then, it is considered to replace it by an **hybrid model**.

This hybrid model will allow meta-data to be exchanged between various editors within the STP project. It is designed to be flexible and extensible. The hybrid model approach would displace the SCA as being the core, although each project could optionally use it if they wished to exchange data. The hybrid model is to serve as unifying baseline for SOA projects, as it attempts to model not just services, components and bindings, but also other concepts that are part of common SOA modelling and infrastructure technologies. In this respect it is better than the SCA meta model, which is mostly about component construction, composition and policy.

1.2 Contributions

This work package proposes some new tools needed for modeling, designing, and implementing SCA based applications. Some of these tools are contributed as new tools to the STP project, others add new features to existing STP tools, and others are particular to the SCOrWare project.

Proposed tools are classified according to 6 categories:

- **Tools for the common meta model between platform and tooling**. The common meta model between platform and tooling, described in [4] (Chapter 3), needs tools to be represented in memory, to be read, to be written, to be validated, etc. This meta model is the starting point for all our developed SCA tools. Chapter 2 contains specification of these tools. These specifications are related to task 2.1. The section 2.4, that explains how to extend the meta model, was added in comparison with the version 1 of this document.
- **Development tools**. Tools that simplify the SCA developer tasks are important in order for this technology will be adopted by the largest community. While graphical designer and other tools mainly target end-user and components assembler, it is clear that developers also need tools to be more efficient. These tools that are related to task 2.2 are described in Chapter 3: (1) wizards that simplify the development of new SCA components, (2) two editors that help the creation of SCA assembly files, (3) a tool for Java developers to add SCA annotations in Java code. Compared with the version 1 of the specifications, this chapter was fully updated.
- **Graphical designers**. These tools target end-users and component assembler. These tools that are related to task 2.3 are described in Chapter 4. The proposed graphical designer provides both bottom-up and top-down methods for constructing graphically SCA composites. Tools related to Business Process are also specified. The section about the SCA Composite Designer extension mechanism is new (Section 4.1). Sections 4.2, 4.3 and 4.4 was improved.
- **Test and deployment tools**. These tools concern test and deployment of SCA based applications. First, it is proposed to integrate the SCOrWare runtime in Eclipse to test SCA based applications. Second, a graphical designer is specified. It provides a quick and easy way to construct graphically the deployment plan of SCA applications. This graphical designer can generate deployment configuration files that can be used by the tool proposed in [4] (Chapter 8 Section 2 - SCA System Deployment with FDF). These tools, related to task 2.4, are described in Chapter 5. Sections 5.2 and 5.3 were fully updated.
- **Tools for searching and semantic composition of services** (Chapter 6). These tools help, developers and integrators that build on the SCOrWare platform, finding and better reusing SCA services and components. This chapter contains specifications related to task 2.5. This chapter was fully updated.
- **Tools for choreography of process that use generic services** (Chapter 7). The objective of these tools is to obtain a workflow and orchestration solution that is able to work with generic services. These solutions must be enough flexible to respect development time and runtime requirements. This chapter contains specifications related to task 2.6. Sections 7.1, 7.2 and 7.3 were updated.

Chapter 8 concludes this document.

2 Common meta model between platform and tooling

This Chapter concerns task 2.1. The purpose of this task is to develop tools to manipulate the common meta model for the SCOrWare runtime and the SCA tools. Section 2.1 presents the objectives, Section 2.2 the specifications, Section 2.3 the implementation and the Section 2.4 explains how to extend the SCA meta model.

2.1 Objectives

To develop tools to manipulate the common meta model we take advantage of the Eclipse Modeling Framework (EMF) [5]. EMF is a modeling framework and code generation facility to build tools and other applications based on a structured data model. From a model specification, described in XMI, EMF provides tools and runtime support to produce:

- The model as a set of Java classes and
- A set of adapter classes that enable viewing and command-based editing of the model.

2.2 Specification

EMF needs only the SCA meta model defined in [4] (Chapter 3) to build automatically desired tools. In Figure 2 is shown the diagram representing the SCA meta model.

Meta model additions

Some existing links between elements in the XSDs are not represented by references. For example, the SCA meta model element *Service* has a field *promote* that links it with a *ComponentService.* Tools need to know explicitly that a reference exists between these two elements.

Then, for each field representing a link between SCA elements, we add a reference as depicted in Figure 1. These new references are not saved but calculated from the field representing the link. These added references appear in Figure 2.

SCA Element	Field representing a link	Reference added
Service	The promote field identifies the promoted S*ervice*	promote2 that references the promoted *ComponentService*
Reference	The promote field identifies the promoted *Reference*	promote2 that references the promoted *ComponentReference*
Wire	The source field names the source component reference	source2 that references the promoted *ComponentReference*
Wire	The target field names the target component service	target2 that references the promoted *ComponentService*

Figure 1: Meta Model Additions

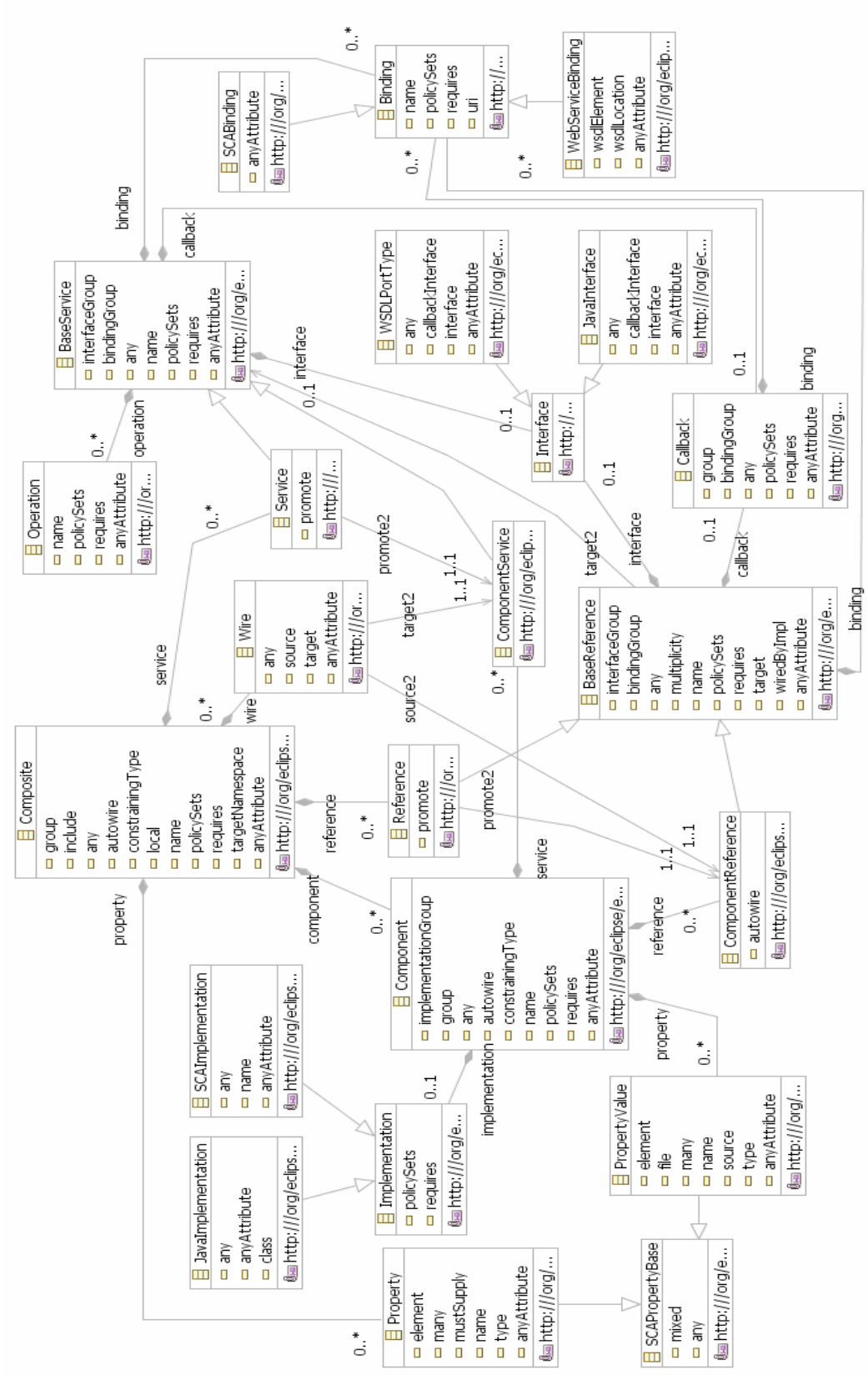


Figure 2: SCA meta model

Additional validation rules

EMF allows to add additional validation rules that are not managed by the meta model. In fact, we need to add additional validation rules that appear in the SCA specification documents but that are not considered by the XSD scheme shown in [4] (Chapter 3). We distinguish 2 types of rules:

- The rules which apply on the SCA assembly description files and
- The rules which verify coherence between the SCA assembly description files and the implementation.

In the following are presented rules found in the SCA specification document [6] (the number between parenthesis is the line number of the rule in the specification document)

Rules which apply on the SCA assembly description files

- Uniqueness of names:
  - The component name must be unique across all the components in the composite (L142).
  - The name of a composite reference must be unique across all the composite references in the composite (L1325).
  - The name of a composite service must be unique across all the composite services in the composite (L1498).
  - The name attribute allows distinction between multiple binding elements on a single service or reference. The default value of the name attribute is the service or reference name. When a service or reference has multiple bindings, only one can have the default value; all others must have a value specified that is unique within the service or reference (L2319) .
- Multiplicity:
  - Multiple target services (target is an attribute of a service) are valid when the reference has a multiplicity of 0..n or 1..n. (L1581) (L203) (L211).
  - The value specified for the multiplicity attribute of a composite reference has to be compatible with the multiplicity specified on the component reference, i.e. it has to be equal or further restrict (L1364).
  - The target attribute of a composite reference is a list of one or more target services, depending on multiplicity setting (L1342).
  - The same component reference may be promoted more than once, using different composite references, but only if the multiplicity defined on the component reference is 0..n or 1..n. The multiplicity on the composite reference can restrict accordingly (L1392).
- Visibility:
  - For a composite used as a component implementation, wires can only link sources and targets that are contained in the same composite (irrespective of which file or files are used to describe the composite). Wiring to entities outside the composite is done through services and references of the composite with wiring defined by the next higher composite (L1630).
  - Components within the composite cannot be referenced directly by the using component. The using component can only connect wires to the services and references of the used composite and set values for any properties of the composite. The internal construction of the composite is invisible to the using component (L1852).
  - Composite services involve the promotion of one service of one of the components within the composite (L1059).
  - Composite references involve the promotion of one or more references of one or more components (within the composite) (L1061).
  - When a component is constrained by a constrainingType (via the "constrainingType" attribute), the entire componentType associated with the component and its implementation is not visible to the

containing composite. The containing composite can only see a projection of the componentType associated with the component and implementation as scoped by the constrainingType of the component. For example, an additional service provided by the implementation which is not in the constrainingType associated with the component cannot be promoted by the containing composite (L2188).

- Valid URI:
  - Valid URI schemes are <component-name>/<reference-name> for the attribute source of a wire. The specification of the reference name is optional if the source component only has one reference (L1621).
  - Valid URI schemes are <component-name>/<service-name> for the attribute target of a wire. The specification of the service name is optional if the target component only has one service with a compatible interface (L1625) .
  - The value of the promote attribute of a composite reference is a list of values of the form <component-name>/<reference-name> separated by spaces. The specification of the reference name is optional if the component has only one reference (L1328).
  - The promote attribute value of a composite service is of the form <component-name>/<service-name>. The service name is optional if the target component only has one service (L1501).
- Validity:
  - A composite used as a component implementation must honor a completeness contract. The concept of completeness of the composite implies that (L1857):
    - the composite must have at least one service or at least one reference.
    - each service offered by the composite must be wired to a service of a component or to a composite reference
  - Two or more component references may be promoted by one composite reference, but only when (L1395):
    - the interfaces of the component references are the same, or if the composite reference itself declares an interface then all the component references must have interfaces which are compatible with the composite reference interface
    - the multiplicities of the component references are compatible, i.e one can be the restricted form of the another, which also means that the composite reference carries the restricted form either implicitly or explicitly
    - the intents declared on the component references must be compatible – the intents which apply to the composite reference in this case are the union of the required intents specified for each of the promoted component references. If any intents contradict (e.g. mutually incompatible qualifiers for a particular intent) then there is an error.
  - If the attribute “WiredByImpl” of a reference is set to true, then the reference should not be wired statically within a composite, but left unwired (L220).
  - If the attribute “WiredByImpl” of a composite reference is set to true, then the reference should not be wired statically within a using composite,but left unwired (L1351).
  - uri attribute of a binding is optional for references defined in composites used as component implementations, but required for references defined in composites contributed to SCA domains (L2307).

Rules which verify coherence between the SCA assembly description files and the implementation

- Name and type matching with implementation:
  - The name of a service has to match a name of a service defined by the implementation (L164).

- The name of a reference has to match a name of a reference defined by the implementation (L192)
- The name of the property has to match a name of a property defined by the implementation (L279). The property type specified must be compatible with the type of the property declared by the implementation (L274).

- Compatible interface:
  - If an interface is specified for a service, it must provide a compatible subset of the interface provided by the implementation (L177).
  - If an interface is specified for a reference, it must provide a compatible subset of the interface provided by the implementation (L226).
  - If an interface is specified for a service composite it must be the same or a compatible subset of the interface provided by the promoted component service, i.e. provide a subset of the operations defined by the component service (L1512).
  - If an interface for a composite reference is specified it must provide an interface which is the same or which is a compatible superset of the interface declared by the promoted component reference (L1358).
  - A wire may only connect a source to a target if the target implements an interface that is compatible (see document) with the interface required by the source (L1634).
  - A wire can connect between different interface languages (e.g. Java interfaces and WSDL portTypes) in either direction, as long as the operations defined by the two interface types are equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and faults/exceptions map to each other (L1648).
- Validity:
  - When an implementation is constrained by a constrainingType it must define all the services, references and properties specified in the corresponding constrainingType. The constraining type's references and services will have interfaces specified and may have intents specified. An implementation may contain additional services, additional optional references and additional optional properties, but cannot contain additional non-optional references or additional non optional properties (a non-optional property is one with no default value applied) (L2182).
  - The constrainingType can include required intents on any element. Those intents are applied to any component that uses that constrainingType. In other words, if requires="reliability" exists on a constrainingType, or its child service or reference elements, then a constrained component or its implementation must include requires="reliability" on the component or implementation or on its corresponding service or reference. Note that the component or implementation may use a qualified form of an intent specified in unqualified form in the constrainingType, but if the constrainingType uses the qualified form, then the component or implementation must also use the qualified form, otherwise there is an error (L2196).
  - The rule which forbids mixing of wires specified with the target attribute with the specification of endpoints in binding sub elements of the reference also applies to wires specified via separate wire elements (L1592).

Meta model extensibility

The core SCA meta model, defined above, can be extended by addition of new implementation, interface and binding types.

Thus, element types defined by FraSCAti (JBI binding, Fractal implementation) and Tuscany (RMI binding, Xquery implementation, ...) are defined by a specific meta model. The user can also defines its own element types in a specific meta model that extends the SCA core meta model.

2.3 Implementation

To build the meta model code and related tools (parser, editor, ...), we use the EMF framework.

Generate ecore meta model from XSD

First, to define the meta model we use the EMF framework to generate it from the XSD files. In the properties view, we can adjust the properties of the meta model elements.

Code generation

Second, to do useful work with the generated meta model, we have to generate a number of artifacts. First of all, we need to generate the ***genmodel*** (Figure 3 (a)). This is basically a decorator model around our meta model that "decorates" a number of extra properties on top of it. Once we have generated the *genmodel*, we select the root element, and execute “Generate All” from the context menu. This will generate (Figure 3 (b)):

- The **implementation classes for the meta model**. Each meta model element has a corresponding interface and implementation. Each generated interface contains getter and setter methods for each attribute and reference of the corresponding model class. Each generated implementation class includes implementations of the getters and setters defined in the corresponding interface, plus some other methods required by the EMF framework. The generated get method simply returns an instance variable representing the attribute. The set method, although a little more complicated, needs to send change notification to any observers that may be listening to the object. The generated accessors for references, especially two-way ones, are a little more complicated and start to show the real value of the EMF generator. See [7] for more explanations about the generated code.
- The **.edit project** that contains a number of generic reusable classes for building editors. It provides:
  - Content and label provider classes, property source support, and other convenience classes that allow EMF models to be displayed using standard desktop (JFace) viewers and property sheets.
  - A command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo.
  - A code generator capable of generating everything needed to build a complete editor plug-in. We can then customize the code however we like, without losing our connection to the model.

  See [8] for more explanations about the generated code.
- The **.editor project** that contains a tree-based editor, specific to that meta model. After running this plug-in, we can define instances of our meta model with a tree view.
- Finally, the **.test project** that contains a number of tests for our meta model.

Additional validation rules

Third, we add additional validation rules with the **framework EMF validation** [9] to check these additional constrains. The EMF validation framework provides a mean to evaluate and ensure the well-formedness of EMF models.

Validation comes in two forms: batch and live. While batch validation allows a client to explicitly evaluate a group of *EObjects* and their contents, live validation can be used by clients to listen on change notifications to *EObjects* to immediately verify that the change does not violate the well-formedness of the model. Client contexts can be specified to ensure that certain constraints do not get executed outside of a certain context. A context is defined by the client to set up their own boundaries from other clients.

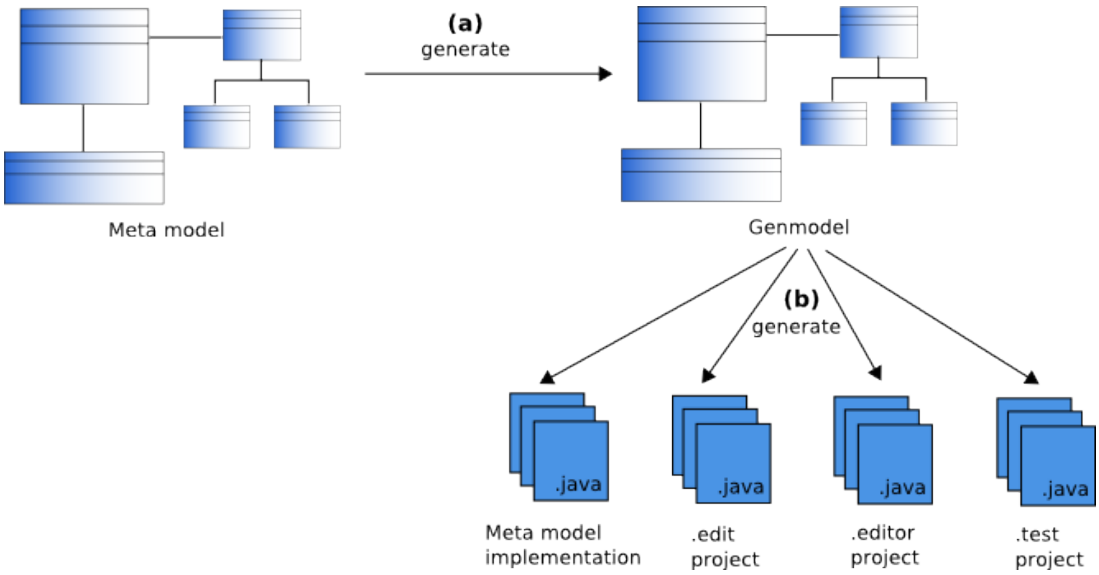


Figure 3: Use of EMF Framework

Meta model extensions provided by SCA Tools

A meta model extension for Tuscany and another for FraSCAti are provided by the SCA Tools project.

Figure 4shows the FraSCAti meta model defined:

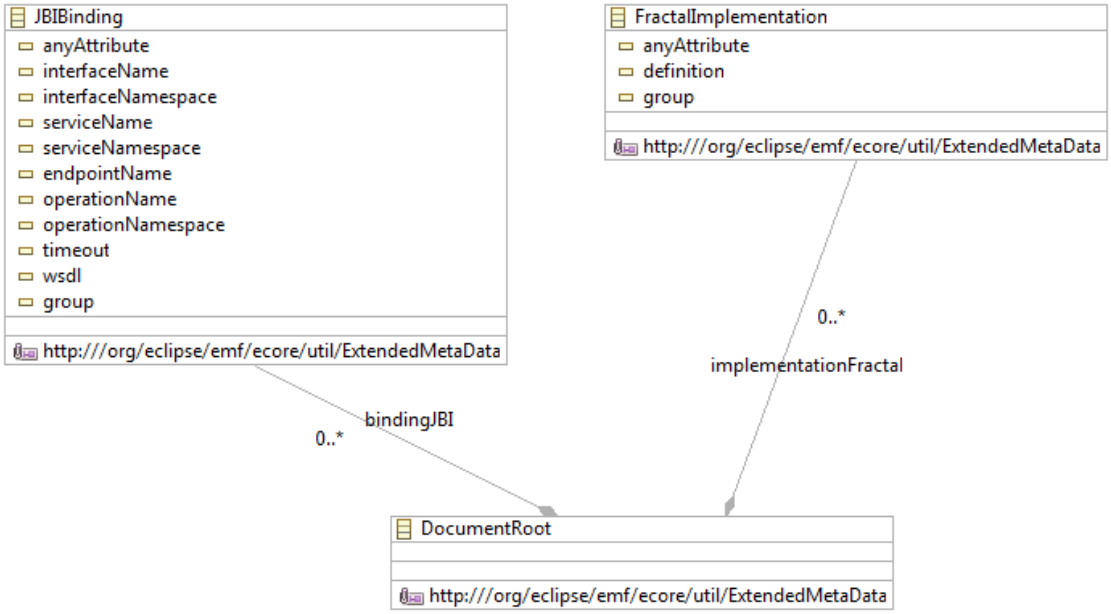


Figure 4: FraSCAti meta model

Figure 5 represents the Tuscany meta model:

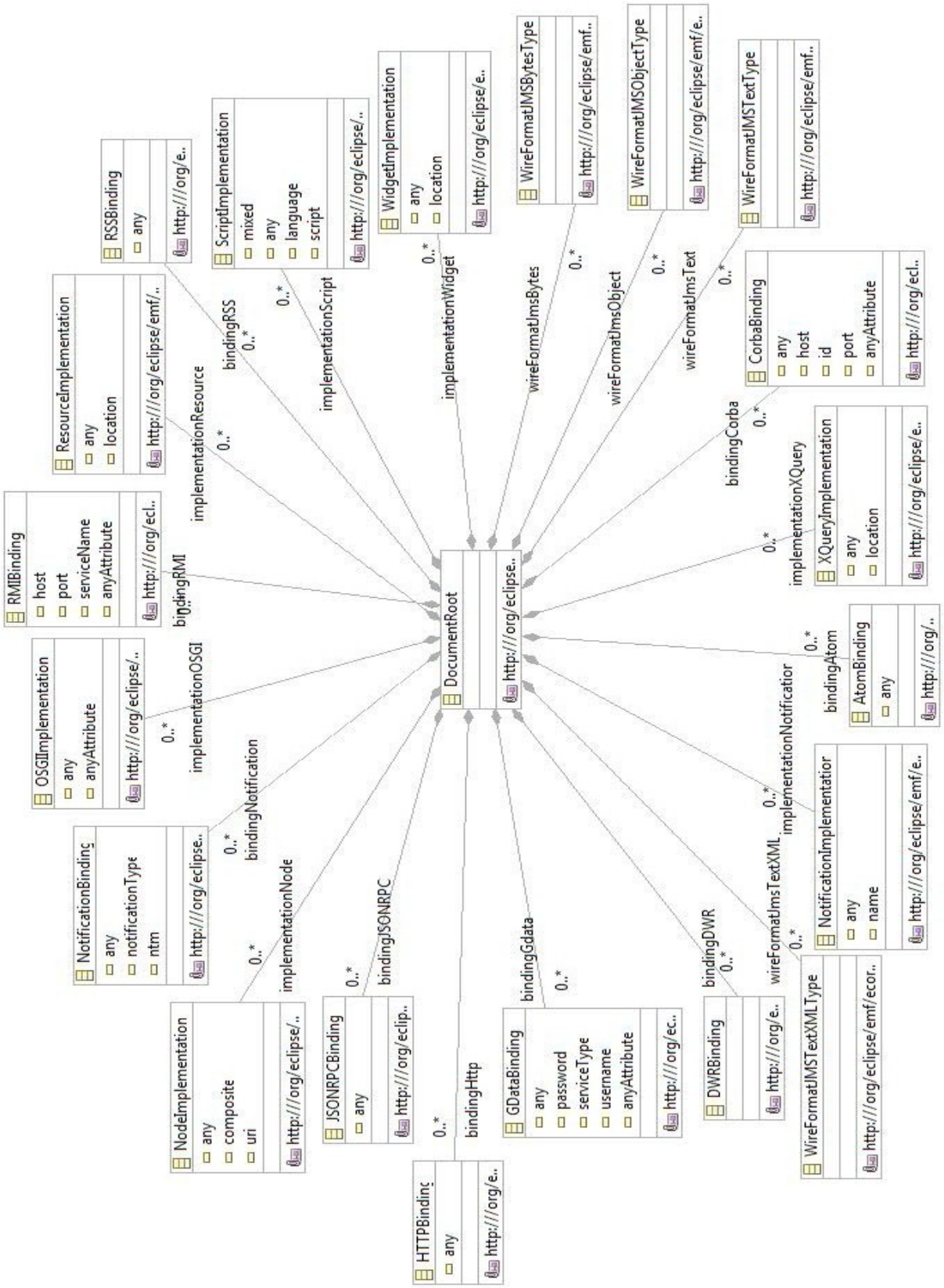


Figure 5: Tuscany meta model

2.4 How to extend the SCA meta model?

This section explains, step by step, how to extend the SCA meta model.

2.4.1 Create a new plugin

First, create a new plugin: **File > New > Project...**, then select **Plug-in Project** (Figure 6). Click **Next**. Set the project name (Figure 7).

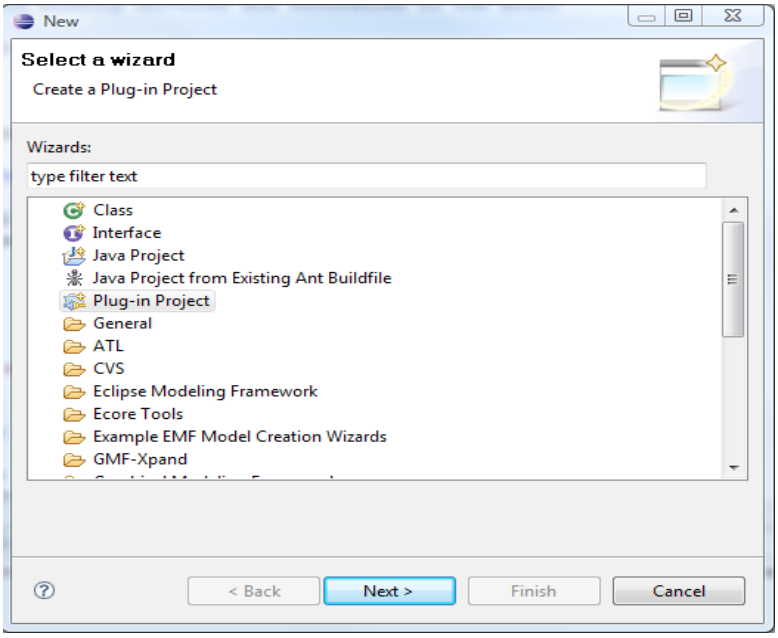


Figure 6: Plug-in Project wizard

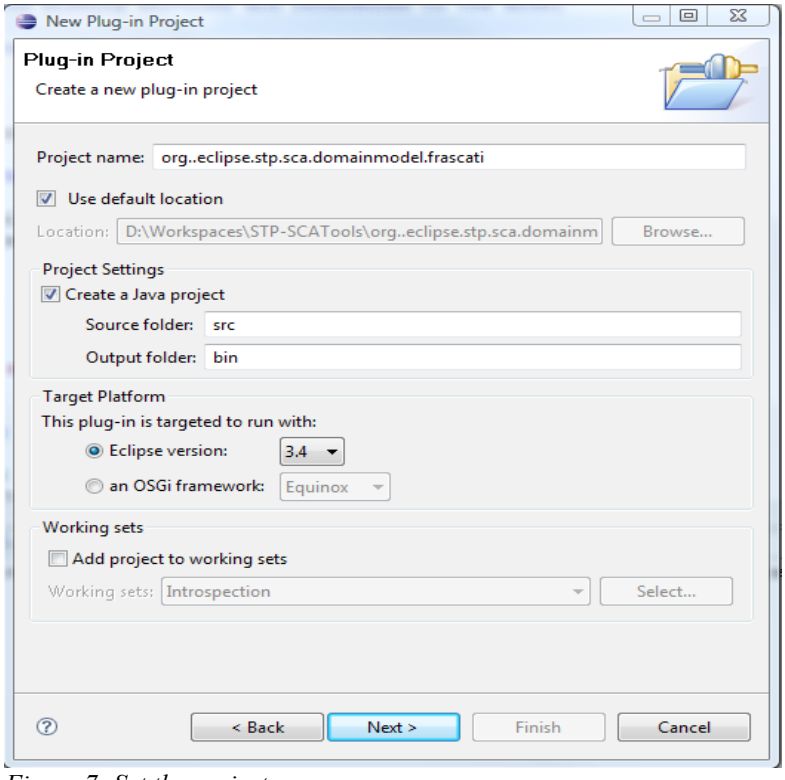


Figure 7: Set the project name

Create a directory named “model” in the new created project (Figure 8).

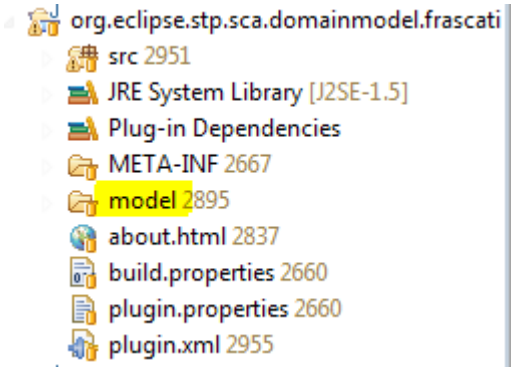


Figure 8: Model directory

2.4.2 Create an SCA meta model extension

Create a new Ecore model (Figure 9).

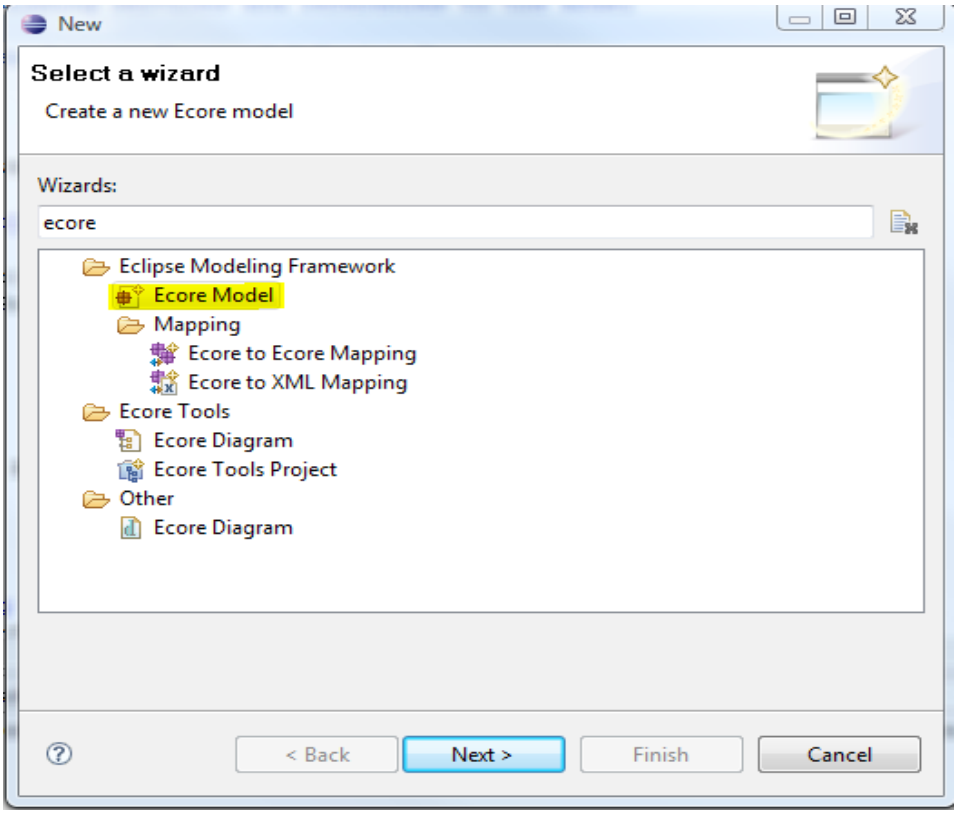


Figure 9: Ecore creation wizard

Open the ecore model, then load the SCA meta model (**right click > Load Resource...**). Click **Browse Registered Package**, select the SCA/OSOA meta model (Figure 10).

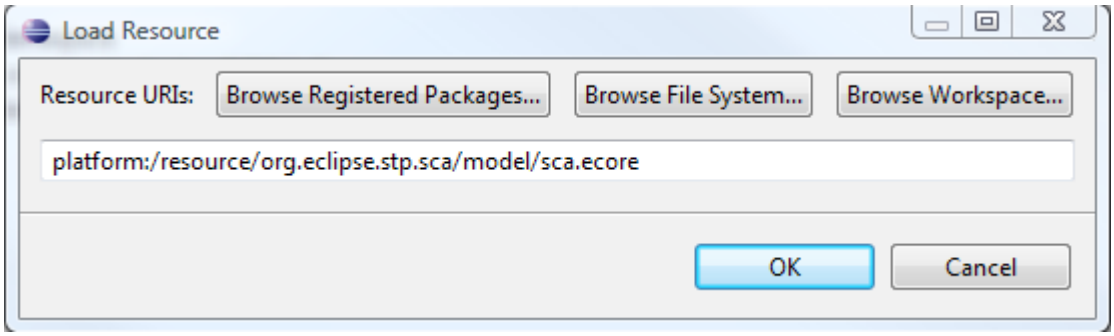


Figure 10: Load the SCA/OSOA meta model

Select the EPackage of your meta model, set the Name, NSPrefix and the NSURI of your meta model.

Add a new EClass named DocumentRoot (Figure 11). This EClass must extend the DocumentRoot of the SCA meta model. Add a new EAnnotation (source = http:///org/eclipse/emf/ecore/util/ExtendedMetaData).

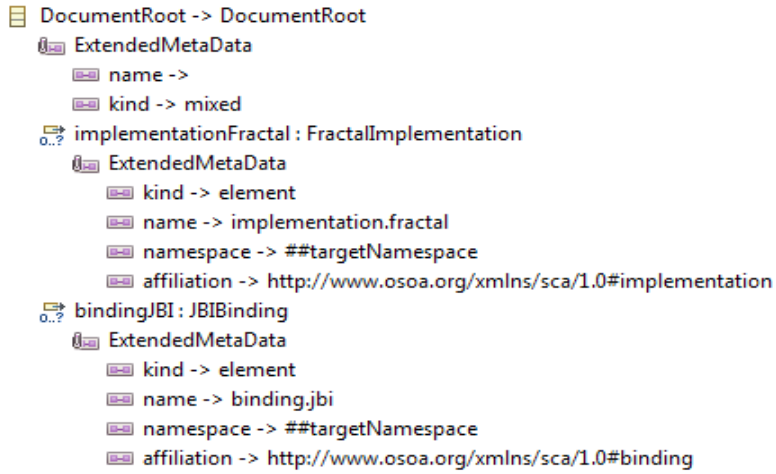


Figure 11: New DocumentRoot

Add twotDetailEntry:

- the first one : name → empty Value.
- the second one: kind → mixed.

Then, under the DocumentRoot EClass add a new EReference. Set the name, the cardinality (min:0, max: -2), and the containment (true). Then, add a new EAnnotation with the following DetailEntry:

- kind → element
- name → scaType.name (for instance implementation.fractal or binding.jbi)
- namespace → ##targetNamespace
- affiliation→http://www.osoa.org/xmlns/sca/1.0#scaType (scaType = binding, interface or implementation).

2.4.3 Create a new SCA Element

Select the EPackage and create a new EClass. Set the name (for instance FractalImplementation), the ESuper Type (Implementation, Interface or Binding).

Create a new EAnnotation (Figure 12), with the following DetailEntry:

- name → SCA Type name

– kind → elementOnly

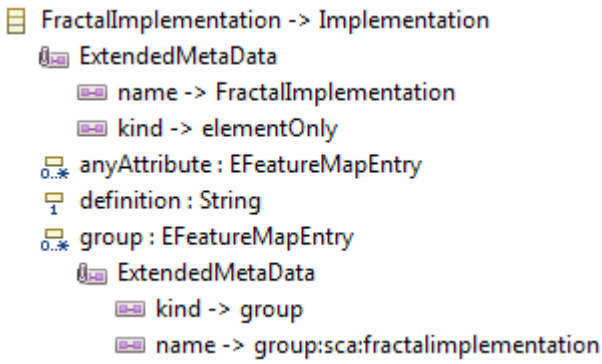


Figure 12: New SCA Element

Then, add the attributes and references for this new SCA element.

2.4.4 Generate the meta model code

Create a new genmodel: **File>New>Other>EMF Model** (Figure 13).

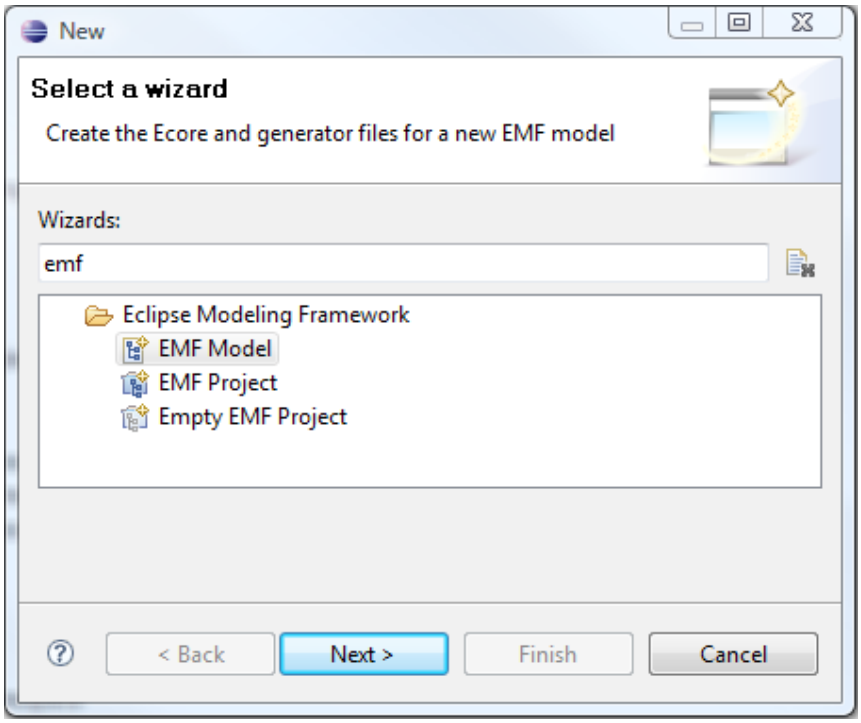


Figure 13: EMF Model wizard

Click the **Next** button. Set the genmodel name, click the **Next** button. Select **Ecore model** as Model Importer (Figure 14), click the **Next** button**.**

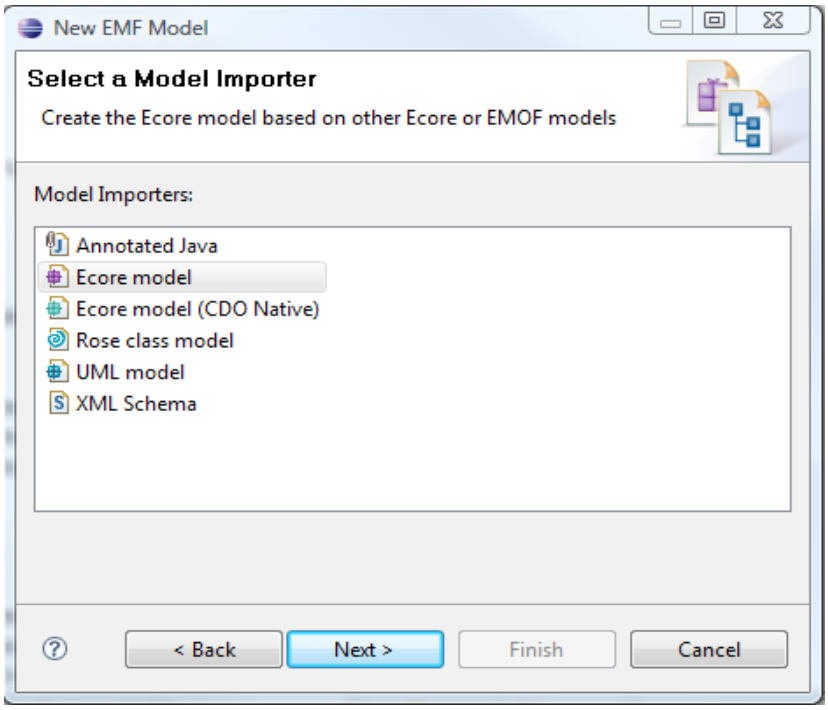


Figure 14: Select a Model Importer

Select the ecore file defined in the previous section (Figure 15), then click the **Next** button. Select which package to generate (Figure 16) and the referenced generator models (sca, sca.policy and sca.instance). Click the **Finish** button. Then, generate the code.

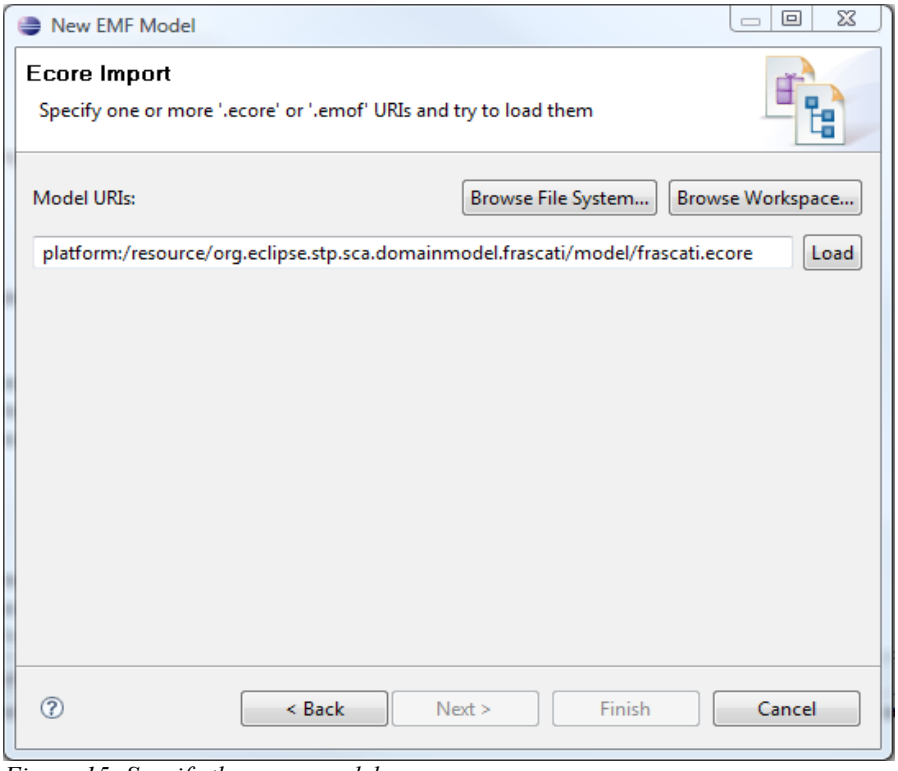


Figure 15: Specify the ecore model

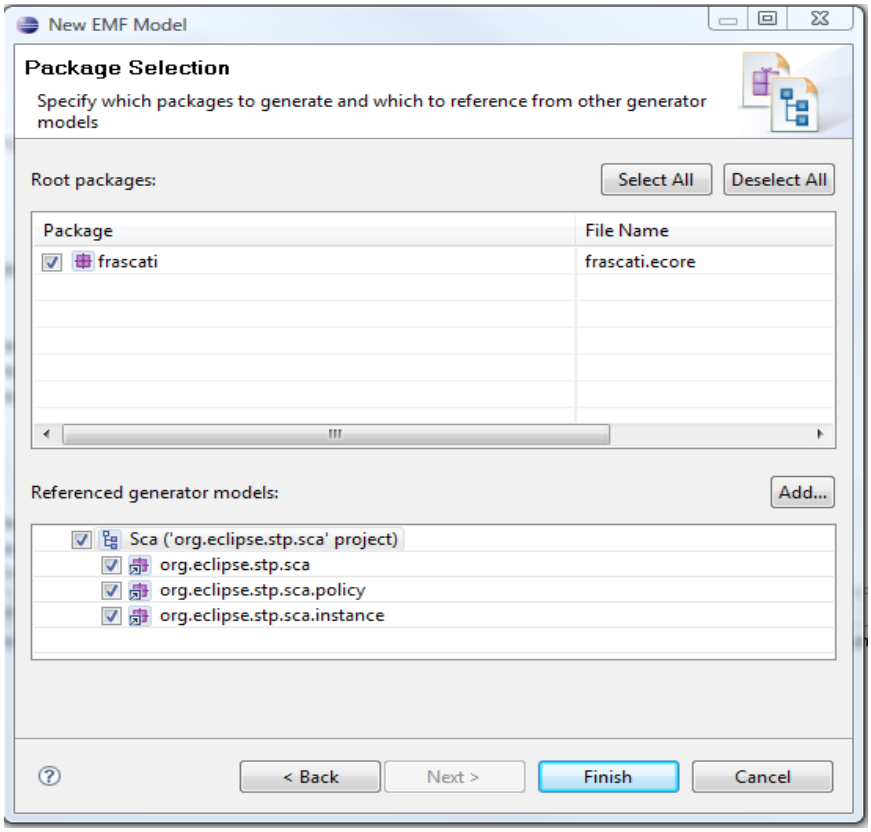


Figure 16: Specify which package to generate

3 Development tools

The task 2.2 aims at providing basic Eclipse tooling for SCA developers. These tools bring help to:

- Create and manage SCA projects in the user workspace (Section 3.1).
- Define and validate SCA assemblies (Section 3.2).
- Facilitate the development of Java implementations (Section 3.3).

3.1 SCA Creation Wizards

3.1.1 Create an SCA project

One important aspect of the SCA specification is that an SCA component support several programming languages to define its implementation. This means that if you have a component inside a composite, this component can be implemented in Java, or in C++, in PHP, BPEL...

Besides, a composite may include (and generally has) several components. And each component may be implemented in a different language.

In the scope of the SCOrWare project, only Java implementations will be supported. However, other open-source implantations of the SCA specification, like Apache Tuscany, support many other implementation types.

Since this tooling is not only intended to SCOrWare users, but to SCA users in general, we should try to cover most of the cases with respect to the implementations. Or at least, provide a minimum of flexibility.

SCA project with Java implementations only

The first case is the one covered by the SCOrWare project. It consists in defining SCA composites whose components only have Java implementations.

The wizard is launched by selecting: **File > New > Other > SOA Tools > SCA > SCA Java project**

The wizard is made up of only one page, asking for:

- the project name
- the JRE version
- the container (where to save the file)

When the wizards completes, it creates a Java project with an SCA nature.

A quite similar wizard has already been contributed into the Service Creation (SC) archive-component of Eclipse STP. For the moment, we propose a simple version of the SC wizard. We will see later whether we migrate toward or use this other wizard.

Basic SCA project

The wizard is launched by selecting: **File > New > Other > SOA Tools > SCA > SCA Project**
The wizard is made up of only one page, asking for:

- the project name
- the container (where to save the file)

When the wizards completes, it creates an empty project with an SCA nature.

It is then to the user to manage his implementations in other Eclipse projects and associate them with its composites.

This kind of projects is intended for users willing to separate the management of the composites and the implementations. As an example, let's imagine the following scenario:

1. A user has defined three composites, interacting together using includes and composite implementation mechanism.
2. The implementations he is using are heterogeneous: one composite use Java implementations, another one use a BPEL implementation, and the last one use composite, Java and script implementations.
3. Making only one big project to manage all of these elements will be a hard task. But making one project to manage the composites, another one to manage the Java implementations, a different one to manage the script and BPEL implementations, will ease the job of the user for the development and the maintenance of the application.

Using such a kind of project can be useful and ease the management and the maintenance of the SCA application. It is a way to break big SCA projects into smaller parts, easier to manage separately.

Moreover, this kind of project is probably the kind of project that will be used by SCA developers who will need other implementations than Java. As an example, users who want to implement their components with C++ inside Eclipse will never use (and neither have the idea) to create an SCA project with the "SCA Java Project" wizard. The "SCA Project" wizard is more neutral than the first one and contributes to use Eclipse and its SOA tooling with Java but also with other languages.

One possible extension of this basic project (not foreseen to be done in this document), is the definition of dependencies between projects in the workspace. An SCA project would be able to depend on other projects, e.g. Java projects, to manage implementations. These dependencies would be used during the validation step to make sure the top-composite does not contain any error.

Packaging

These wizards will be proposed to the Eclipse STP project to integrate the SCA sub-project as soon as they reach a stable version. License: EPL.

3.1.2 Create an SCA composite

This wizard is launched by selecting: **File > New > Other > SOA Tools > SCA > Composite File**

The wizard is made up of only one page, asking for:

- the composite file name
- the container (where to save the file)

When the wizards completes, it creates a composite file with the given name. The skeleton of the file is also created. Eventually, this file is opened in the default editor.

3.1.3 Create an SCA componentType

This wizard is launched by selecting: **File > New > Other > SOA Tools > SCA > Component Type**

The wizard is made up of only one page, asking for:

- the component type file name
- the container (where to save the file)

When the wizards completes, it creates a componentType file with the given name. The skeleton of the file is also created. Eventually, this file is opened in the default editor.

When the wizard is launched after the selection of a composite file, the wizard page will be pre-filled with the composite name.

This wizard will be proposed to the Eclipse STP project to integrate the SCA sub-project as soon as it reaches a stable version. License: EPL.

3.2 SCA Editors

3.2.1 XML editor for *.composite files

Composite files are XML files.

As a result, they can be edited in any XML editor. In particular, the XML editor provided by the Eclipse WTP project can be customized to open and edit these files. However, this editor does not take care of the specifics of the SCA specification.

We propose to extend this editor to avoid these limitations. Thus, this XML editor for *.composite files brings the following features:

- Inherited features
  - Syntax highlighting.
  - Document formatting.
  - Validation as-you-type based on XSD meta-model files.
  - A Property view to edit element attributes.
  - Context assistance (auto-completion for mark-ups and attributes) – based on the XSD files.
- Additional features
  - Composite files automatically associated with this editor.
  - Context assistance / auto-completion for attribute values (Figure 17).
  - Preference page to register elements (bindings, implementations and interfaces) which are not in the meta-model (e.g. Tuscany extra-bindings...).
  - Customized outline view.

Figure 17: What content assistants look like in Eclipse editors

Details about “context assistance for attribute values”:
This context assistance brings auto-completion for attribute values.

More exactly, when the user requires this content assistant by typing in <CTRL> + <Space>, he is proposed choices in the following contexts:

- Attribute “promote” of a composite service: all the services from the contained components are proposed. Proposals take into account the included composites.

Proposals look like ***componentName/serviceName***

- Attribute “promote” of a composite reference: all the references from the contained components are proposed. Proposals take into account the included composites.

Proposals look like ***componentName/referenceName***

- Attribute “target” of a component reference: all the services from the contained components are proposed. Proposals take into account the included composites.

Proposals look like ***componentName/serviceName***

- Attributes “source” and “target” of a wire element. All the services and references from the contained components are proposed. Proposals take into account the included composites.

Proposals look like ***componentName/serviceName*** for the “target” attribute and ***componentName/referenceName*** for the “source” attribute.

- Attribute “name” from an include element: all the composite files in the same project, not already included, are proposed.

Proposals look like ***ns:compositeName*** if the target name space of the composite to include is already defined in the top-composite node and associated with the prefix ***ns***, or ***ns:compositeName xmlns:ns=”targetNameSpace”*** otherwise.

To make things clear about this last point, the editor makes sure to reuse the target name space if it was defined. Otherwise, it creates the name space declaration and generates a name space prefix in the include mark-up. In any case, the proposal result is coherent and valid with respect to XML syntax.

Details about the “preference page for extra-elements”:
This preference page is used to define elements that can not be included into the meta-model (e.g. for technical or copyright reasons). The idea is to make a kind of restricted XSD editor to define and associate runtime platforms, name spaces, binding, implementation and interface elements.

As a preference page, any user can add new SCA elements which are not in the meta-model. As tooling provider, we propose an initial set of predefined elements.

Figure 18 shows the elements that can be registered in this page and the way they are related (constraints do not appear).

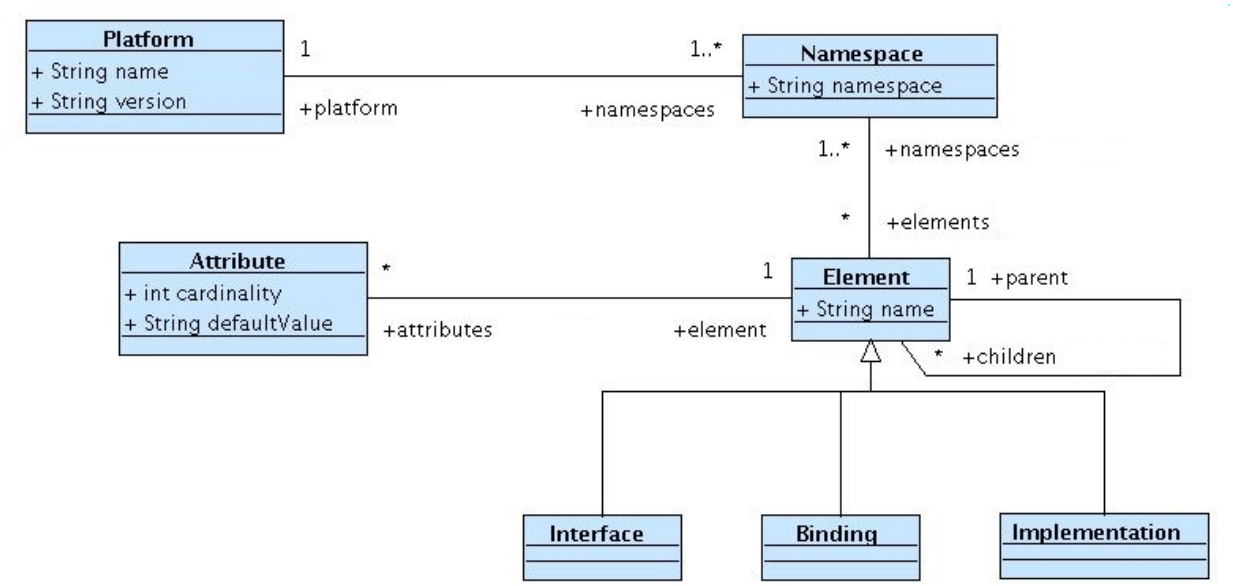


Figure 18: The class diagram for SCA extra-elements defined by users

The preference page defines an easy way to manage these definitions and their dependencies. A tree viewer seems to be the best way to define the XML hierarchy of elements and attributes.

It should allow to export and load definitions from files (so that you can keep and reuse your definitions from one Eclipse to another) and restore initial definitions. Such a page is expected to ease the extensibility of the editors and avoid copyright issues with XSD files.

In the scope of this XML editor, the preference page is used by a custom content assistant which provides auto-completion for bindings, implementations and interfaces with these registered elements.

More exactly, when the user requires this content assistant by typing in <CTRL> + <Space>, he is proposed additional choices in the following contexts (“additional” with respect to the proposals already made from the XSD meta-model files):

- Implementations of a component (implementation types and associated attributes).
- Bindings of a service and a reference.
- Interfaces of a service and a reference.

The proposals made are related to the name spaces defined in the composite node. Only the elements associated with a declared name space are proposed.

This content assistant is integrated in the editor so that the user can not determine from which content assistant (the default one or this one) the proposals come from. To achieve that, the proposals are gathered and then ordered alphabetically before being displayed to the user.

Example of expected use, Tuscany extra-elements:

- Implementations of a component (implementation types and associated attributes).
  - Mark-up “implementation.script” and the attributes “language” and “script”.
  - Mark-up “implementation.spring”.
  - Mark-up “implementation.resource”.
  - Mark-up “implementation.bpel” and the attribute “process”.
  - Mark-up “implementation.xquery” and the attribute “location”.
  - Mark-up “implementation.widget” and the attribute “location”.
  - Mark-up “implementation.osgi” and the attributes “bundleSymbolicName”, “bundleVersion”, “classes” and “imports”. This mark-up also support the sub-mark-up “properties” with the attributes “service”, “reference”, “serviceCallback” and “referenceCallback”.
- Bindings of a service and a reference.
  - Mark-up “binding.ajax”.
  - Mark-up “binding.jms” with the attribute “uri”.
  - Mark-up “binding.jsonrpc”.
  - Mark-up “binding.rmi” with the attributes “host”, “port”, serviceName”.
  - Mark-up “binding.ejb” with the attribute “uri”.
  - Mark-up “binding.rss”.
  - Mark-up “binding.atom”.
- Interfaces of a service and a reference.
  - Mark-up “interface.cpp”.

The attributes and mark-ups documentation comes from Apache Tuscany's website. The mark-ups whose attributes are not specified may have attributes, but they were not documented. Since this is only a content assistant and given that these elements are not all stable, these lacks are not a major problem. This will be updated when these elements are known.

As explained before, these proposals are made only when the Tuscany name space is defined in the document. To provide some more help to the user, this content assistant proposes the Tuscany name space (http://apache.tuscany.org/xmlns/sca/1.0) when the user requires content assistance inside an “xmlns” attribute. If this name space is not defined, the proposals given above are not made.

Details about “the customized outline view”:
The outline view provided by the XML editor from the Eclipse WTP project is an outline view for XML documents.

It does not manage the specific aspects of SCA, like inclusions, and the icons do not help to distinguish the elements.

The XML editor for *.composite defines an outline view specific to SCA composites. A set of icons, defined in the SCOrWare project, are shared among editors and designers. These icons are reused in this outline view. Only SCA elements appear in this outline view (no XML instruction or XML comment).

The viewer used in the outline is a tree viewer (Figure 19). Its root is the composite element. Its children are the children elements. The apparition order of these children is pre-defined:

1. components
2. services
3. references
4. wires
5. properties
6. includes

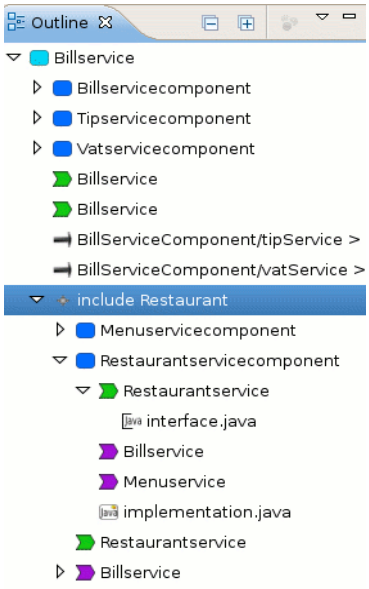


Figure 19: Overview of what the outline view should look like

Since an include element means “include a composite”, an include element in the outline view has a sub-tree as unique child. This sub-tree is in fact the elements of the composite, displayed in the order given above.

Every node has a set of actions the user can select.

Thus, all the nodes have a “show properties” action, which opens the Eclipse properties view and shows the details about this node. Include elements have an “open file” action, which opens the included file in the default editor.

Select a node in the outline view also selects the node in the editor, and vice-versa (in the same way that what is provided by the default XML editor from WTP).

About composite inclusions, it is not clear, at the moment, how runtime platforms handle them (for Tucany as well as FraSCAti). Consequently, recursive inclusions are not managed by this editor. That is to say that if a composite A includes another composite B, which itself also includes another composite C, then the editor, when it makes proposals for composite A, does not proposes elements from composite C.

This editor is bundled as a distinct plug-in for the Eclipse plat-form, and will be proposed to the Eclipse STP project to integrate the SCA sub-project as soon as it reaches a stable version. License: EPL.

3.2.2 Form editor for *.composite files

A second editor is proposed for SCA developers. This one should be seen as an intermediate editor between the XML editor (which is a text editor) and the SCA graphical designer. Based on Eclipse forms, this multi-page editor allows you to edit a *.composite file through a Web form interface, more user-friendly than a text editor but less than a GMF-based editor. A typical example of such an editor is the PDE editor of the Eclipse platform (the editor which edits plugin.xml files when developing Eclipse plug-ins).

SCA elements are here managed separately, each main concept being edited in a separate page. An important usage is made of lists (lists of components, services, references, properties, includes), and this is why this editor is the most adapted to composite having a lot of elements.

As it was said, this editor is a multi-page editor (Figure 20). It has the following pages:

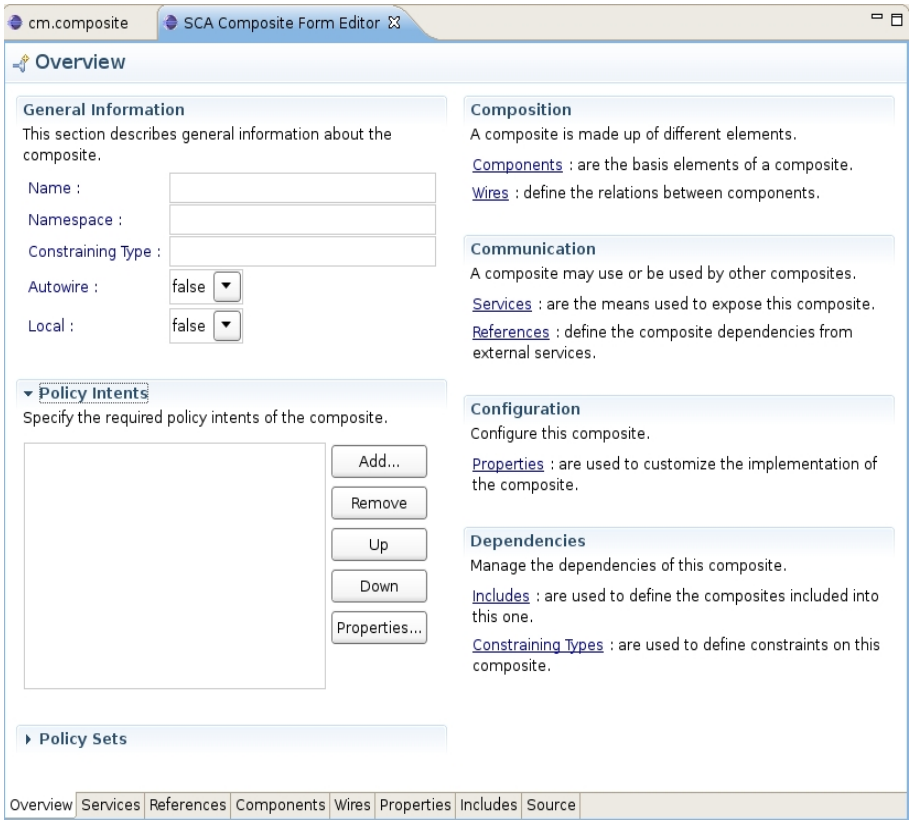


*Figure 20: Overview of the form editor for *.composite files*

- Overview: gives an overview of the composite (name, target name space, constraining type, requires, intents) and links to other pages.
- Components: lists the components and their elements (services, references properties, implementation).
- Services: lists the services of the composite.
- References: lists the references of the composite.
- Wires: lists the wires of the composite (explicit wires), the references whose “target” attributes are set (implicit wires) and the components whose “autowire” attribute is set to true (auto-wired).
- Constraining Types: edit the constraining types defined by a composite.
- Includes: lists the composite files included into this composite.
- Source: uses the XML editor for *.composite files proposed in the previous section to display the source of the composite.

All the pages share a same model. As a result, they are synchronized, including the source page.

As described in Eclipse User Interface Guidelines, the outline view for this editor respects the way the pages are organized. Thus, the organization of the pages inside the editor is also found into the outline view. To ensure coherence with the other SCA tools described in this document, the icons used in the editor and in its outline view are the same than those used by the XML editor and the graphical designer for *.composite files.

From the maintainability point of view, it is planned to reuse the preference page described in section **1.2.1** to extend the binding, implementation and interface editing widgets. By using this page, this editor dynamically creates the required graphical elements to add or modify these extra-elements.

It is not clear, at the moment, whether this editor will provide or not a property view. In both cases, this is not really important since the editor allows you to edit any element attribute.

This editor is bundled as a distinct plug-in for the Eclipse plat-form, and will be proposed to the Eclipse STP project to integrate the SCA sub-project as soon as it reaches a stable version. License: EPL.

3.2.3 XML editor for *.componentType files

Component Types are SCA elements used to set implementation details that then do not have to be added in the composite anymore. They are defined in a file with the extension ***.**componentType. A component type uses a subset of the elements of a composite. Its schema is also defined in the XSD files used by the XML editor for *.composite files.

Consequently, the XML editor described in 1.1.1 can be reused for *.componentType files.

The only modifications to perform are the following:

- Associate *.componentType files with this editor.
- Update the outline view to allow componentType elements as root.

To conclude, the editor proposed in 1.1 will be used to edit both *.composite and *.componentType files.

3.3 Convenience Tools for Developers

3.3.1 Eclipse builder for SCA projects

The SCOrWare project proposes several tools to edit and manipulate SCA elements. This is true in particular for *.composite files, since there are two editors and one graphical designer to modify them.

One missing complementary aspect to edition is validation.

When Java developers work with Eclipse, they benefit from the Java validation to indicate them when they typed in a wrong symbol or when they wrote something incorrect. This is the kind of features developers appreciate and that we propose here for SCA elements.

A validation module will be used by developers when they define their application to validate the artifacts before deploying and executing them. This is why this validation part is divided into two parts:

- The validation core: performs the validation itself. It can be used by both the tooling and the platforms.
- The integration inside the tooling: it is achieved by using a builder which validates the project each time one of its resources is modified. Each error, warning and information is associated with a file marker indicating in the editors what elements are concerned and why.

The validation core

The validation is made up of several steps.

1. The project validation step.

   Validate the project itself. These rules cannot be checked with the EMF meta-model but are important, in

particular for the management of composite inclusions.

- Ensure that for every composite file, the composite file name is the same than the composite name.

For every *compositeName.composite* file, the composite mark-up of the file has an attribute name whose value is *compositeName*.

- Ensure that inside a project, the couple (targetNamespace, compositeName) is unique.

c1.composite and *c2.composite* two files of a same SCA project implies (targetNamespace of c1, c1) != (targetNamespace of c2, c2)

2. The completion step.

   This step consists in loading the meta-model instance from the composite and completing it with information retrieved from the implementations. A completion module is expected to be provided in the Eclipse STP project by an actor external to the SCOrWare project. It may also receive contributions from the work package 1. If this module comes too late, it will not be integrated in the validation core and only the composites will be used to create the instance of the meta-model. Seen from the validation point of view, it is not considered as a validation step but as a part of the next step.

3. The EMF validation.

   This step consists in validating the meta-model instance built in the previous step. This validation is made with respect to the meta-model and its constraints defined in the section 1 of this document.

4. Other validations.

   The last step is used to validate other SCA elements, e.g. policies, respect of constraining types. Nothing is foreseen to be implemented for this part. It is mentioned as a possible extension.

The project validation step can not work in the same way in Eclipse (whose file system is based on the notion of "resource") and in the platform (whose file system is the usual one). The validation core has to provide a solution to that.

For now, nothing is sure about the completion code. If it is contributed in the Eclipse STP project, nothing indicates this code will be reusable by the platforms. This point will be discussed when a clear proposal about this part shows up.

The EMF validation part can be used without any problem by both parties.

The policy adopted when an error is found during a validation step is to terminate this step, skip the next steps, and returned the errors as EMF diagnostics. EMF diagnostics provide a powerful and common way to describe errors within meta-model instances. We propose to make it the standard object to express the result of a validation in this module.

Integration within Eclipse

As said before, the integration of the validation module is made through a project builder.

The idea is to associate this builder with every created SCA project. This association is ensured by defining and using an SCA nature. This SCA nature was mentioned in the sections describing the wizards to create an SCA project.

When an SCA project is created, it is automatically associated with the SCA nature (and in consequence, associated with the builder). Each time a resource (a file, a folder) is modified in the project (update, creation, deletion), the builder is called and performs a validation of the project by calling the validation module (detailed in the previous section).

The validation returns a list of EMF diagnostics, which are then used to refresh the problem markers on the files. More exactly, before any validation, all the markers are removed. Then, the validation is called and new markers are

created if necessary. These markers are visible in the editors for *.composite and *.composite_diagram files.

An Eclipse marker is a mechanism provided by the platform. A marker is always associated with a resource. It is used to indicate something to the user. In general, markers are used to show errors.

Adding a marker like we need on a resource has three consequences:

- An overlay icon is displayed on the file icon.
- A message is displayed in the “problems” view.
- An icon may be displayed in the editors using this file as input.

Diagnostics will be used to determine whether markers need to be added. This is achieved by checking the diagnostic level of every diagnostic.

- Severity < Information Level => IMarker.SEVERITY_INFO
- Severity < Warning level => IMarker.SEVERITY_WARNING
- Otherwise => IMarker.SEVERITY_ERROR

The main difficulty in this part is the marker synchronization in the validation of *.composite files. Indeed, there are two ways to edit a composite. You can edit it directly by using an editor, or you can edit by using the SCA Designer. This second edition kind is indirect because the designer edit both the *.composite file and a *.composite_diagram file.

What the user sees in the designer is the *.composite_diagram file. Not directly the composite. Consequently, adding markers in the designer requires the addition of markers on the *.composite_diagram file while the validation core adds markers on the *.composite file. So, we would need to add markers on both kind of files. But if we do that, the messages in the “problems” view appear twice (one message from each file), which is not coherent and convenient for the user.

What has to be done is to add the markers on the *.composite file, and add the associated images on the diagram without adding markers on the associated composite_diagram file. This implies we have to by-pass some GMF mechanisms.

Packaging

The SCA builder will be proposed to the Eclipse STP project to integrate the SCA sub-project as soon as it reaches a stable version. License: EPL and maybe another license (double-licensing) to make sure the platforms will be able to reuse the code.

3.3.2 SCA annotations for Java implementations

Java is the only supported language for implementations in the SCOrWare project. To improve and facilitate the development of Java implementations for SCA, we propose to provide the Java annotations for SCA in the Eclipse Java editor.

Since Apache Tuscany already proposed a framework with annotations in the Eclipse STP project, we plan to study which annotations are missing from this tool and extend it so that all the annotations from the SCOrWare project are integrated in the SCA tooling.

Besides, we have to try to move this annotation mechanism from the Service Creation (SC) component of STP to the SCA component. Our contribution will be proposed to the Eclipse STP project to integrate the SCA sub-project as soon as it reaches a stable version. License: EPL.

4 Graphical designers

This section presents graphical tools for Eclipse (Task 2.3) that simplify the SCA developer task. Section 4.1 specifies the SCA composite designer. This graphical editor helps the development and the configuration of SCA assemblies. Section 4.2 introduces Business Process (BP) tooling and integration in SCOrWare. Section 4.3 presents the JWT business designer, and Section 4.4 the JWT technical designer.

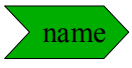
4.1 SCA Composite designer

4.1.1 Objectives

The SCA Composite editor is a graphical tool to construct SCA composite files. The tool will provide both bottom-up and top-down methods for constructing standard SCA 1.0 composites. This graphical editor will respect the graphical representation proposed in the SCA specifications. This Eclipse tool will be developed with the GMF framework and from the SCA meta model.

Graphical representation of a component

For some SCA artifacts (service, reference, property, component, composite, wire, promote) the specification document [6] proposes a graphical representation. For the SCA artifacts not represented in the specification (implementation, interface and binding) we propose our own graphical representation. The SCA document specification depicts:

- A service by the following green figure: 

. The service name is on the top of the figure. The same figure is used for a component service and for a composite service. The only difference is in the properties where a composite service has a property “promote”.
- A reference by the following purple figure: 

. The reference name must be on the figure. The same figure is used for a component reference and for a composite reference. The only difference is in the properties where a composite reference has a property “promote”.
- A property by a yellow rectangle: 

. The property name must be on the figure. The same figure is used for a component property and for a service property.
- A component by a blue rounded rectangle (Figure 21). The component name must be at the center of the component. Component services straddle the left side of the component, component references straddle the right side of the component and properties straddle the up of the component.

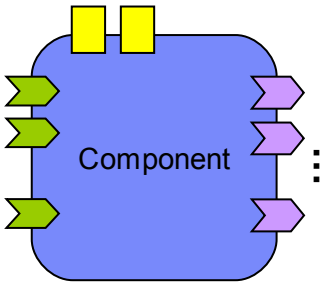


Figure 21: Graphical representation of an SCA component

- A composite by a light blue rounded rectangle (Figure 22). The composite name must be on the left corner of the composite graphical representation. Components are represented inside the composite. Composite services straddle the left side of the composite, composite references straddle the right side of the composite and properties straddle the up of the composite.

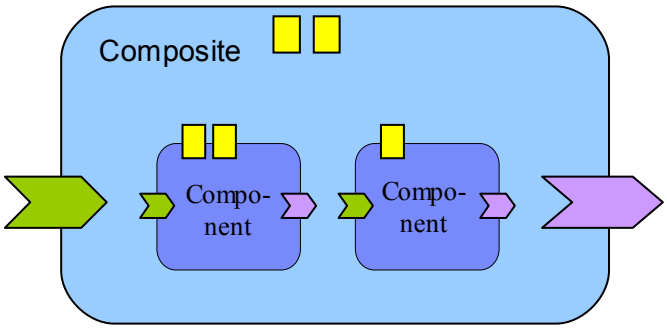


Figure 22: Graphical representation of an SCA composite

- A wire by a black line. Two types of wire are possible: the first one is a wire object, and the second one uses the target property of a reference to set the wire. Wires can only connect a component reference to a component service.
- A promote link by a black dash line. A promote link can be added between a composite service and a component service or between a composite reference and a component reference.

The specification document does not propose a graphical representation for:

- A binding. We propose to represent this SCA element by a red circle on a component/composite service/reference. Bindings can be only added to a service or a reference (of a composite or a component). Several bindings can be added to the same service/reference.
- An implementation. We propose to represent this SCA element by a yellow circle on a component. An implementation can be only associated with a component and only one implementation can be associated to a specific component.
- An interface. We propose to represent this SCA element by an orange circle on a component/composite service/reference. Interfaces can be only added to a service or a reference (of a composite or a component). Only one interface can be added to the same service/reference.

4.1.2 Specification

The SCA composite designer must allow to construct SCA composite configuration files in a top-down manner and in a bottom-up manner.

Top-Down method

This tool works in a *top-down* manner, allowing the creation of composites first, and then the generation of model code.

The process to construct a composite configuration is the following:

- First, a wizard is used to create a new composite configuration. The user picks-up a name for the composite file.
- The SCA composite editor is open with the graphical representation of the new created composite.
- The user can add SCA elements (component, service, reference, wire, binding, implementation, interface, ...). For this, creation tools are available at the right side of the editor in the creation tools palette.

- Interface, implementation and binding can be added also by drag and drop of existing files in the workspace.
- A right-click on an implementation, binding or interface representation proposes a menu item that allows to open the element with the corresponding editor. For example right-click on a BPEL implementation open the file with the BPEL editor.
- A right-click on a component service (or a component reference) proposes a menu item that allows to promote it automatically. For this, the corresponding composite service (or composite reference), its name (the same as the promoted element) and the promote link are created.

Bottom-up method

This tool works also in a *bottom-up* fashion, discovering components that have been developed in code and producing a graphical representation. The process to construct a composite configuration by introspection of the workspace is the following:

- First, like in the top-down approach the wizard allows to pick-up a name for the composite file.
- The graphical editor is open with the newly created composite.
- Then, a right click on the composite representation proposes a menu item to import components from the workspace.
- A new wizard is open. This wizard (like in Figure 23) proposes to select components to be created out of existing implementations. Then, the graphical representation of selected components are created inside the composite. Each component is created with Service, Reference, Implementation and Interface according to information found by introspection of selected components.

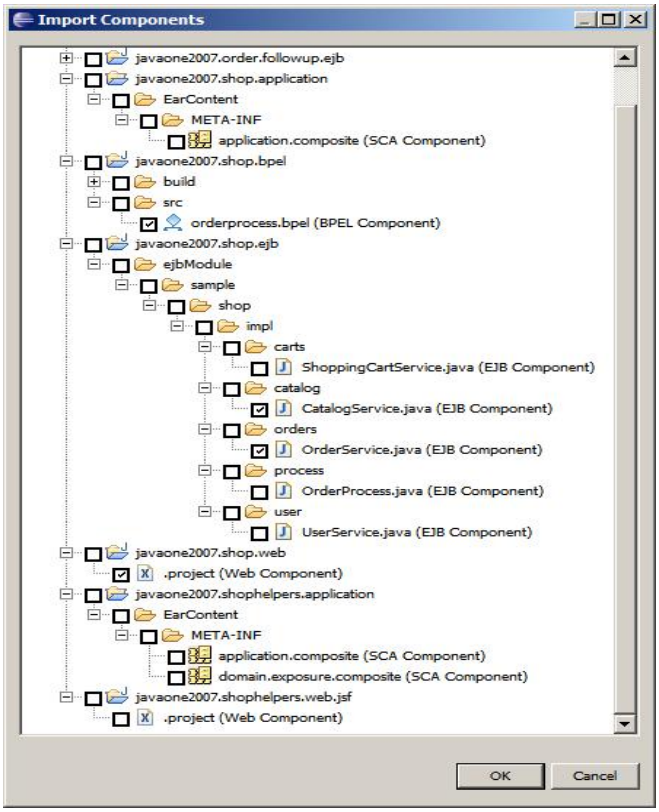


Figure 23: Wizard to select existing components to add to a new SCA composite

- After that, properties on the components, wire and promote links can be added.
- Like in the top-down method:
  - A right-click on an implementation, binding or interface representation proposes a menu item that allows to open the element with the corresponding editor and
  - A right-click on a component service (or a component reference) proposes a menu item that allows to promote it automatically.

Use semantic broking service

This tools will be integrated with the tools developed in Chapter 6 to pick-up services corresponding to a certain set of semantic criteria.

Meta-model

The meta model used to construct this tool is a subset of the SCA meta model + the Tuscany meta model + the FraSCAti meta model. The subset of the SCA meta model contains all elements relative to the *Composite* element.

SCA Composite Designer extension points

To extend the SCA Composite Designer, we define the following extension points:

Identifier: **org.eclipse.stp.sca.diagram.bindings**

Description:

Used to define the factory that is used to provide the interfaces needed to support Viewers.

Configuration Markup:

<!ELEMENT extension (adaptorFactory+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** - The extension point name, `org.eclipse.stp.sca.diagram.AdaptorFactories.`.
- **id** - The extension ID.
- **name** - The extension name.

<!ELEMENT adaptorFactory EMPTY>

<!ATTLIST adaptorFactory class CDATA #REQUIRED>

- **class** - A fully qualified name of the Java class implementing the factory that is used to provide the interfaces needed to support Viewers.

Examples:

```
<extension point="org.eclipse.stp.sca.diagram.AdaptorFactories">
  <adaptorFactory
class="org.eclipse.stp.sca.domainmodel.frascati.provider.FrascatiItemProviderAdapterFactory" />
</extension>
```

Identifier:

org.eclipse.stp.sca.diagram.palette.BindingEntryPalette

Description:

Used to define a new SCA binding to add in the palette.

Configuration Markup:

<!ELEMENT extension (bindingEntry)*>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** - The extension point name, `org.eclipse.stp.sca.diagram.BindingEntryPalette.`.
- **id** - The extension ID.
- **name** - The extension name.

<!ELEMENT bindingEntry EMPTY>

<!ATTLIST bindingEntry

type CDATA #REQUIRED

label CDATA #REQUIRED

description CDATA #REQUIRED

iconPath CDATA #IMPLIED>

The metamodel element type to add in the palette (SCA Binding).

- **type** - The metamodel element type (SCA binding).
- **label** - The display name in the palette for this metamodel element type.
- **description** - The description for this metamodel element type.
- **iconPath** - The path of this metamodel element type icon, relative to this plugin location.

Examples:

```
<extension point="org.eclipse.stp.sca.diagram.BindingEntryPalette">
  <bindingEntry
    description="Create a new JBIBinding"
    iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/JBIBinding.gif"
    label="JBI (FraSCAti 0.4)"
    type="org.eclipse.stp.sca.diagram.frascati.JBIBinding">
  </bindingEntry>
</extension>
```

Identifier:

org.eclipse.stp.sca.diagram.palette.ImplementationEntryPalette

Description:

Used to define a new SCA implementation to add in the palette.

Configuration Markup:

<!ELEMENT extension (implementationEntry)*>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** - The extension point name, `org.eclipse.stp.sca.diagram.ImplementationEntryPalette.`.
- **id** - The extension ID.
- **name** - The extension name.

<!ELEMENT implementationEntry EMPTY>

<!ATTLIST implementationEntry

type CDATA #REQUIRED

label CDATA #REQUIRED

description CDATA #REQUIRED

iconPath CDATA #IMPLIED>

The metamodel element type to add in the palette (SCA Implementation).

- **type** - The metamodel element type (SCA implementation).
- **label** - The display name in the palette for this metamodel element type.
- **description** - The description for this metamodel element type.
- **iconPath** - The path of this metamodel element type icon, relative to this plugin location.

Examples:

```
<extension point="org.eclipse.stp.sca.diagram.ImplementationEntryPalette">
  <implementationEntry
    description="Create a new FractalImplementation"
    iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/FractalImplementa
tion.gif"
    label="Fractal (FraSCAti 0.4)"
    type="org.eclipse.stp.sca.diagram.frascati.FractalImplementation">
  </implementationEntry>
</extension>
```

Identifier:

org.eclipse.stp.sca.diagram.palette.InterfaceEntryPalette

Description:

Used to define a new SCA interface to add in the palette.

Configuration Markup:

<!ELEMENT extension (interfaceEntry)*>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** - The extension point name, `org.eclipse.stp.sca.diagram.InterfaceEntryPalette.`.
- **id** - The extension ID.
- **name** - The extension name.

<!ELEMENT interfaceEntry EMPTY>

<!ATTLIST interfaceEntry

type CDATA #REQUIRED

label CDATA #REQUIRED

description CDATA #REQUIRED

iconPath CDATA #IMPLIED>

The metamodel element type to add in the palette (SCA Interface).

- **type** - The metamodel element type (SCA interface).
- **label** - The display name in the palette for this metamodel element type.
- **description** - The description for this metamodel element type.
- **iconPath** - The path of this metamodel element type icon, relative to this plugin location.

Examples:

```
<extension point="org.eclipse.stp.sca.diagram.InterfaceEntryPalette">
  <bindingEntry
    description="Create a new FooInterface"
    iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/FooInterface.gif"
    label="Foo (FraSCAti 0.4)"
    type="org.eclipse.stp.sca.diagram.frascati.FooInterface">
  </bindingEntry>
</extension>
```

Identifier:

org.eclipse.stp.sca.diagram.binding.bindings

Description:

Used to define the visual representation of a new binding.

Configuration Markup:

<!ELEMENT extension (element*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** - The extension point name, `org.eclipse.stp.sca.diagram.extension.bindings`.
- **id** - The extension ID.
- **name** - The extension name.

<!ELEMENT element EMPTY>

<!ATTLIST element

typeId CDATA #REQUIRED

literalField CDATA #REQUIRED

iconPath CDATA #IMPLIED

literalClass CDATA #REQUIRED>

- **typeId** - The Id of the corresponding metamodel element type (SCA binding).
- **literalField** - The fully qualified name of the class that defines literals for the meta objects that represent
  - each class,
  - each feature of each class,
  - each enum,
  - and each data type
- **iconPath** - The path of this metamodel element type icon, relative to this plugin location.
- **literalClass** - The meta object literal for the corresponding metamodel element type (SCA binding).

Examples:

```
<extension point="org.eclipse.stp.sca.diagram.extension.bindings">
  <element
    typeId="org.eclipse.stp.sca.diagram.frascati.JBIBinding"
    literalClass="org.eclipse.stp.sca.domainmodel.frascati.FrascatiPackage$Literals"
    literalField="DOCUMENT_ROOT__BINDING_JBI"
    iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/JBIBinding.gif" /
>
</extension>
```

Identifier:

org.eclipse.stp.sca.diagram.interface.interfaces

Description:

Used to define the visual representation of a new implementation.

Configuration Markup:

<!ELEMENT extension (element*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** - The extension point name, `org.eclipse.stp.sca.diagram.extension.implementations.`
- **id** - The extension ID.
- **name** - The extension name.

<!ELEMENT element EMPTY>

<!ATTLIST element

typeId CDATA #REQUIRED

literalField CDATA #REQUIRED

iconPath CDATA #IMPLIED

literalClass CDATA #REQUIRED>

- **typeId** - The Id of the corresponding metamodel element type (SCA implementation).
- **literalField** - The fully qualified name of the class that defines literals for the meta objects that represent
  - each class,
  - each feature of each class,
  - each enum,
  - and each data type
- **iconPath** - The path of this metamodel element type icon, relative to this plugin location.
- **literalClass** - The meta object literal for the corresponding metamodel element type (SCA implementation).

Examples:

```
<extension point="org.eclipse.stp.sca.diagram.extension.implementations">
  <element
    typeId="org.eclipse.stp.sca.diagram.frascati.FractalImplementation"
    literalClass="org.eclipse.stp.sca.domainmodel.frascati.FrascatiPackage$Literals"
    literalField="DOCUMENT_ROOT__IMPLEMENTATION_FRACTAL"
    iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/FractalImplementa
tion.gif" />
</extension>
```

Identifier:

org.eclipse.stp.sca.diagram.interface.interfaces

Description:

Used to define the visual representation of a new interface.

Configuration Markup:

<!ELEMENT extension (element*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** - The extension point name, `org.eclipse.stp.sca.diagram.extension.interfaces.`.
- **id** - The extension ID.
- **name** - The extension name.

<!ELEMENT element EMPTY>

<!ATTLIST element

typeId CDATA #REQUIRED

literalField CDATA #REQUIRED

iconPath CDATA #IMPLIED

literalClass CDATA #REQUIRED>

- **typeId** - The Id of the corresponding metamodel element type (SCA interface).
- **literalField** - The fully qualified name of the class that defines literals for the meta objects that represent
  - each class,
  - each feature of each class,
  - each enum,
  - and each data type
- **iconPath** - The path of this metamodel element type icon, relative to this plugin location.
- **literalClass** - The meta object literal for the corresponding metamodel element type (SCA interface).

Examples:

```
<extension point="org.eclipse.stp.sca.diagram.extension.interfaces">
  <element
    typeId="org.eclipse.stp.sca.diagram.frascati.FooInterface"
    literalClass="org.eclipse.stp.sca.domainmodel.frascati.FrascatiPackage$Literals"
    literalField="DOCUMENT_ROOT__INTERFACE_FOO"
    iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/FooInterface.gif"
/>
</extension>
```

4.1.3 Implementation

To build this editor we will use the GMF framework [10]. We start from our meta model (the *.ecore* file) and the *.genmodel* defined above in Section 2.3 (Figure 24 (a)). A number of additional models have to be defined:

- A **model defining the graphical notation** (Figure 24 (c)) including shapes, decorations and graphical nodes and connections. This is called the *.gmfgraph* model.
- A **model for the editor's palette** (Figure 24 (b)) and other tooling, called the *.gmftool* model.
- A **mapping model** (Figure 24 (d)) that binds these two models to the domain meta model. The two models defined above are technically independent of your domain meta model.

From all of these additional models, GMF creates the ***.gmfgen* model** (Figure 24 (e)) – a "low level" model that the code generator uses as an input, finally creating the .diagram project which contains our desired editor.

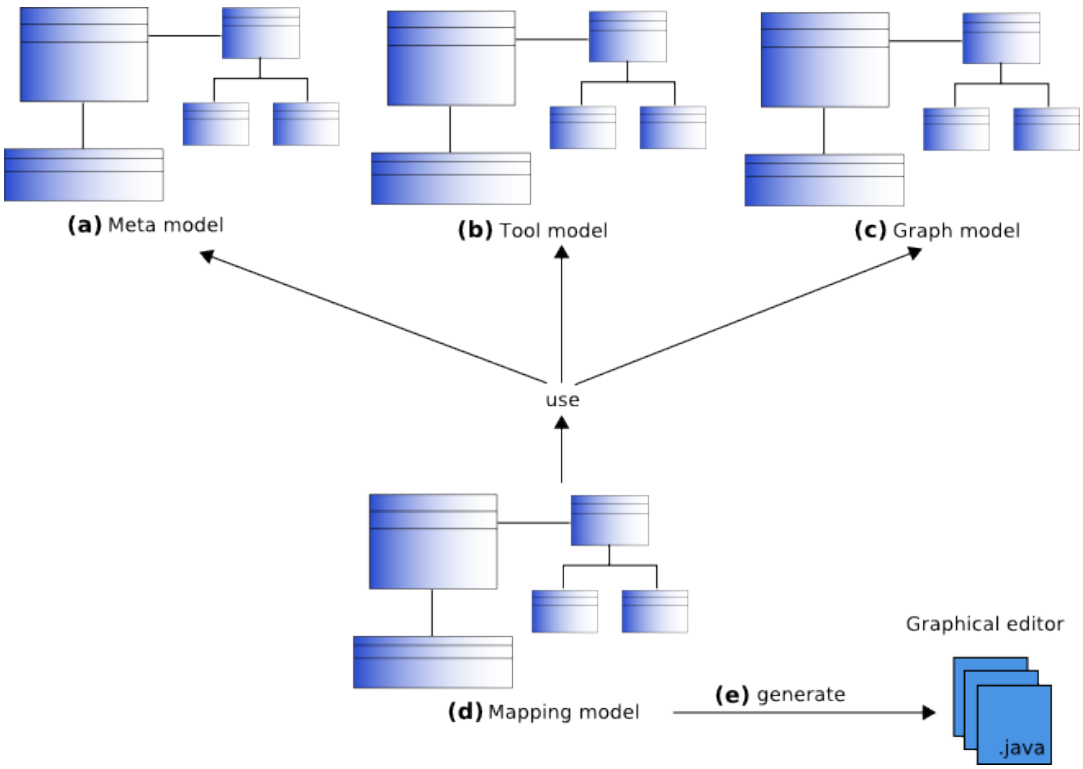


Figure 24: Use of GMF to generate graphical editor

The Tool Model

The *.gmftool* model defines only a set of palette entries. The palette is the set of buttons that allows to add model elements to our model. So we need a creation tool for each of the meta model elements that we want to be able to place onto the editor.

The Graph Model

The graph model defines several things:

- One is the set of figures defined in *FigureGalleries*. Colors, line, and static decorations are also defined here.

- We also define graph nodes and connections.
- We also define compartments. Compartments are sections in nodes that can be collapsed and themselves contain graphs (or lists of elements).
- Finally, we define diagram labels used to show text associated with graphical elements.

The Mapping Model

This is the most complex model. Here we map the tool definition and the graph definition to the domain meta model. To be able to map the different elements, we have to add these other resources to the editor:

- For each meta model element, that we want to map directly onto the diagram surface, we have to define first a Top Node Reference.
- Attached to a Top Node Reference, we add a normal Node Mapping. It contains information about the model element to map and the property, in which the set of these elements is stored in the container.
- Attached to a Node Mapping there's a Label Mapping. This one associates the label defined in the graph with the respective model element properties.
- We can also have a Child Reference that identifies the set of children that should be shown.
- The Child Reference can be associated with the Compartment, to ensure that the child collection is actually shown in the respective compartment.

All of this have to be mapped with a number of properties. The editors for doing that are just the usual tree editors which make all of that stuff a bit cumbersome. There are additional, more specialized editors in the GMF pipeline that should make this process simpler.

Generating the Editor

Now we can create the *.gmfgen* model from the set of models we just create. Finally, from the *.gmfgen* model, we can generate the diagram code – this will be contained in the *.diagram* project.

Specific code must be developed for the service and reference figures. These figures must represent the shapes defined by the SCA specification document. GMF does not take into account natively these types of shape. Moreover, specific code must be developed to represent component and composite according to the specification document. This code must allow to represent services and references on the component and composite border.

Figure 25 shows a prototype of the SCA Composite designer contributed to the STP project by Obeo.

Extension for FraSCAti and Tuscany:

The SCA elements defined by FraSCAti and Tuscany are added to the SCA Composite Designer using the extension points defined above.

The following figure contains the implementation of the extension points for FraSCAti.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

<extension point="org.eclipse.gmf.runtime.emf.type.core.elementTypes">
  <metamodel nsURI="http://frascati.ow2.org">
    <metamodelType
                id="org.eclipse.stp.sca.diagram.frascati.FractalImplementation"
                name="FractalImplementation"
                eclass="FractalImplementation"
                kind="org.eclipse.gmf.runtime.emf.type.core.IHintedType">
      <param
             name="semanticHint"
             value="org.eclipse.stp.sca.diagram.frascati.FractalImplementation"/>
    </metamodelType>
```

```
    <metamodelType
                  id="org.eclipse.stp.sca.diagram.frascati.JBIBinding"
                  name="JBIBinding"
                  eclass="JBIBinding"
                  kind="org.eclipse.gmf.runtime.emf.type.core.IHintedType">
      <param name="semanticHint" value="org.eclipse.stp.sca.diagram.frascati.JBIBinding"/>
    </metamodelType>
  </metamodel>
</extension>

<extension point="org.eclipse.gmf.runtime.emf.type.core.elementTypeBindings">
  <clientContext id="ScaClientContext">
    <enablement>
      <test
           property="org.eclipse.gmf.runtime.emf.core.editingDomain"
           value="org.eclipse.stp.sca.diagram.EditingDomain"/>
    </enablement>
  </clientContext>
  <binding context="ScaClientContext">
    <elementType ref="org.eclipse.stp.sca.diagram.frascati.FractalImplementation"/>
    <advice ref="org.eclipse.gmf.runtime.diagram.core.advice.notationDepdendents"/>
  </binding>
  <binding context="ScaClientContext">
    <elementType ref="org.eclipse.stp.sca.diagram.frascati.JBIBinding"/>
    <advice ref="org.eclipse.gmf.runtime.diagram.core.advice.notationDepdendents"/>
  </binding>
</extension>

<extension point="org.eclipse.stp.sca.diagram.extension.implementations">
  <element
          typeId="org.eclipse.stp.sca.diagram.frascati.FractalImplementation"
          literalClass="org.eclipse.stp.sca.domainmodel.frascati.FrascatiPackage$Literals"
          literalField="DOCUMENT_ROOT__IMPLEMENTATION_FRACTAL"
          iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/FractalImple
mentation.gif"/>
</extension>

<extension point="org.eclipse.stp.sca.diagram.extension.bindings">
  <element
          typeId="org.eclipse.stp.sca.diagram.frascati.JBIBinding"
          literalClass="org.eclipse.stp.sca.domainmodel.frascati.FrascatiPackage$Literals"
          literalField="DOCUMENT_ROOT__BINDING_JBI"
     iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/JBIBinding.gif"/>
</extension>

<extension point="org.eclipse.stp.sca.diagram.AdaptorFactories">
  <adaptorFactory
    class="org.eclipse.stp.sca.domainmodel.frascati.provider.FrascatiItemProviderAdapterFactory"
  />
</extension>

<!-- Palette Provider -->
<extension point="org.eclipse.stp.sca.diagram.ImplementationEntryPalette">
  <implementationEntry
     description="Create a new FractalImplementation"
     iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/FractalImplementa
tion.gif"
     label="Fractal (Frascati 0.4)"
     type="org.eclipse.stp.sca.diagram.frascati.FractalImplementation">
  </implementationEntry>
</extension>

<extension point="org.eclipse.stp.sca.diagram.BindingEntryPalette">
  <bindingEntry
     description="Create a new JBIBinding"
     iconPath="/org.eclipse.stp.sca.domainmodel.frascati.edit/icons/full/obj16/JBIBinding.gif"
     label="JBI (Frascati 0.4)"
     type="org.eclipse.stp.sca.diagram.frascati.JBIBinding">
  </bindingEntry>
</extension>

</plugin>
```

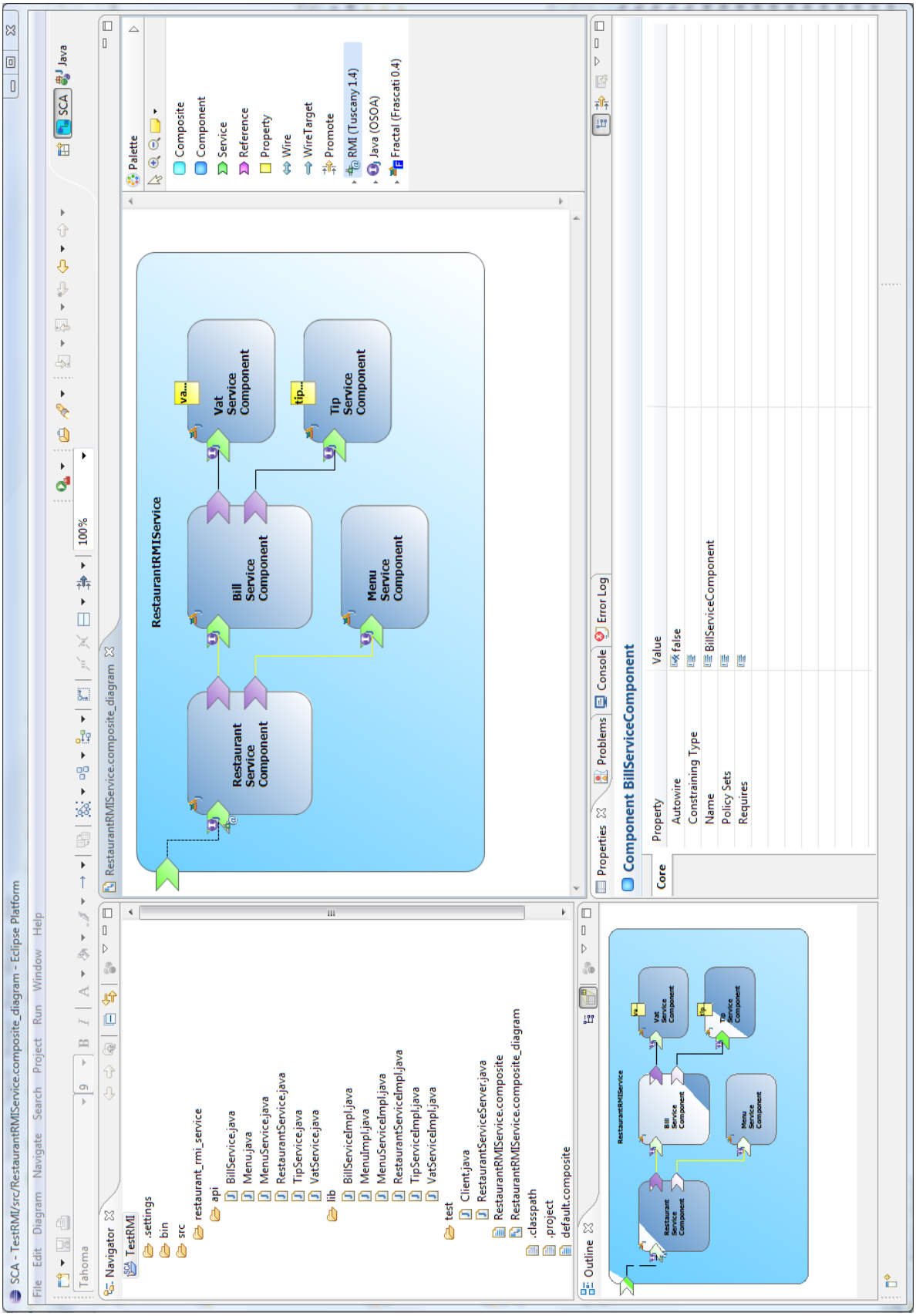


Figure 25: SCA Composite designer

4.2 Preliminary note about BP tooling and integration in SCOrWare

JWT for SOA / SCOrWare tooling suite

JWT's goal in the context of the SCOrWare platform is to provide a top-down, business-oriented approach to modeling business processes on top of a Service Oriented Architecture and implementing business tasks using the SCOrWare SOA framework. More concretely, this includes first and foremost providing a business, high-level process view and providing tooling for building processes using SCOrWare services, but also providing graphical tooling for services within processes by taking advantage of semantic service search and Eclipse STP projects like IM and SCA, and last but not least allowing such SOA processes to be executed, by designing the JWT Runtime APIs and providing their SCOrWare and OW2-based Scarbo implementation.

JWT's larger scope objectives – that is, a unified approach to BPM development tooling – are in line with the SCOrWare specific objectives, since SOA as a methodology and SCA as a technology are in their own way addressing the same kind of problematics, such as : heterogeneity of information systems, heterogeneity of the "business to design to runtime" toolchain, decoupling business and infrastructure and managing them.

Contributions

Service design-time tools are provided to STP and process design-time tools to JWT, save for specific features like SCA service semantic search and resolution that are contributed to Scarbo.

Process runtime developments are contributed to Scarbo.

Target user population

Design-time service tools can be used whatever the service (webservices, RMI or SCA) runtime technology used (ex. Tuscany for the latest). However runtime features depends on the runtime of choice.

Design-time process tools can be used whatever the process runtime technology, provided there is a JWT Transformation exporting to its executable process format (NB. There are already several such transformations, like XPDL, and writing another has been made as simple a task as possible). However SOA within processes is only available at runtime by using a process engine that implements JWT Runtime APIs, like Scarbo's TaskEngineFramework implementation.

Dissemination

Achieving those goals include dissemination of JWT and its SOA / SCOrWare developments. This is done through the JWT and Scarbo communities (website, maling list, newsgroup), their members and integrators (in addition to Open Wide, notably the University of Augsburg with sourceforge's AgilPro, and recently Bull and Red Hat jBoss with the development of their next generation process editor on top of JWT), their partner projects like STP (especially IM and SCA, with which common use cases with JWT have been defined and integrated), OW2 FraSCAti and Bonita projects, interested third-parties like OW2 Spagic, but also to Eclipse events like Eclipse Europe Summit 2008 and Eclipse Conference 2009, and to OW2 events like OW2 TechDays at Open World Forum 2008 and OW2 Conference at Linux Solutions 2009. Our ambitions are embodied in the planned integration of JWT in Eclipse's next yearly release 3.5 "Galileo", with the support of JWT mentor John Graham (former Eclipse DTP top-level project lead).

Busines Process (BP) modeling tooling suite

The workflow and orchestration tool suite is provided by JWT, the Scarbo project and the hereby specified additional developments.

Nowadays JWT is available as a complete workflow solution, featuring a BP designer (its Workflow Editor or WE), using its own BP format and coming along with a standalone simulator application allowing BP execution.

The top-down approach that BP brings to SOA features two different BP design use cases:

- The BP analyst's. Its goal is to describe processes at a high, business level). Its prominent standard is BPMN.
- The BP developer's. Its goal is to implement processes on top of an infrastructure, notably process engines. Standard formats are BPEL, XPDL etc.

In order to support both use cases, as well as increase its standard compliance, it will feature a BPMN compliant business designer on one side, and support a standard BP format on the other side. This will is achieved with the help of JWT's format interoperability (through JWT Transformations) and meta model genericity through JWT metamodel extensions and its Conf framework (which provides EMF Aspects and Profiles).

Other parts of the SCOrWare BP solution

This part is about BP designers.

See further for :

- JWT designer features with SOA and SCOrWare services
- Service / Composite registry (implemented on top of the SCA service / component semantic provider)
- JWT process and service execution integration (Scarbo 's implementation of JWT Runtime APIs)
- Process engine monitoring

High-level description of the BP on SOA tool chain and metamodels

Here is a diagram (Figure 26) showing how the different BP models (including BP representation models, graphical editors meta models, BP language format models) actually interact within the JWT for SCOrWare BP solution, thanks to JWT's key notion of *pivotal meta model* :

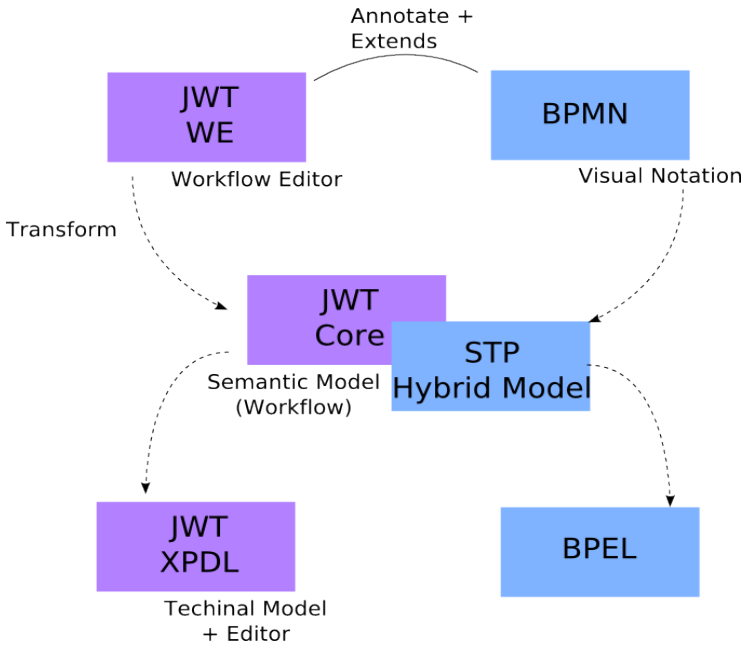


Figure 26: JWT / BPMN / BPEL / STP Hybrid Model

The JWT core meta model acts as a pivotal meta model in that as much as possible every other meta model for which there is a bijective transformation with it has access to the full JWT features, just like the core meta model has

– and therefore both should be able to be transparently swapped. This is showed in the middle line of the diagram.

The upper line of the diagram shows business process representation editors. Those are targeted at business analyst rather than developers. Those editors allows to edit information that may be used, thanks to a transformation (usually one way), as a starting point for more developer oriented work done in the JWT technical editor or on a pivotal meta model.

On the lower line of the diagram stand the technical tools, allowing to produce Bps in their own language format, possibly even targeted at a specific platform, like SCOrWare. Those are produced through another transformation from a pivotal meta model (usually one way), and they may also have their own editor.

Just as JWT as a special place in this toolchain on the process side, STP IM had a special place in it on the service side. Common STP – JWT vision and use cases have been defined, where JWT builds processes on top of services defined in a central STP IM model, allowing them to be managed across STP IM compatible service-oriented tools. In addition, the University of Augsburg has integrated (outside of SCOrWare) JWT and STP IM through JWT Transformations and the model-to-model ATL technology, which allows any tool provided by STP and compatible with STP IM to benefit to the JWT process-on-SOA design toolchain.

4.3 JWT business designer

4.3.1 Objectives

The objective is to develop the tooling required by the BP analyst use case of the top-down approach that BP brings to SOA.

This will be achieved with the help of JWT's format interoperability (JWT Transformations) and meta model genericity (JWT metamodel extensibility and Conf framework, which provides EMF Aspects and Profiles).

It is concretely a BPMN compliant design tool, achieved by integrating with JWT the BPMN Editor provided by the Eclipse STP BPMN project.

4.3.2 Specifications

About BPMN for SCOrWare

BPMN is a graphical notation for modeling business processes. It is meant to have all the expressive power needed for the business analyst to design its business processes. As such a SCOrWare integrated BPMN designer is the business oriented tool of choice in the first step of a top-down approach of BP design in the SCOrWare platform.

However it is important to note that it is not a BP language in itself : for example, it has no text format representation and it can't be executed. Concretely, it talks about shapes and edges with BP related meaning like task, transition and so on ; but there is no implementation information.

The point of a BPMN diagram is what it means, therefore we will have

1. to specify this meaning in the context of SCOrWare (see BPMN subset afterwards)
2. to define a textual (XML) format for ex. exchange with other tools (see EMF meta model)

BPMN subset and meaning

A subset of BPMN that will be supported first will be defined. This subset has to be enough for the needs of the SCOrWare partners and the common uses of SCOrWare. The aim is to already have a fully functional one in year one. This study will be done by Open Wide with the help of Obeo, IRIT and the JWT community.

This subset will be able to represent

- activity (task),
- action and implementation specific extensions (service call etc.),

- guarded transitions (decisions),
- loops,
- business role,
- data.

BPMN graphical modeling tool

The aim is to provide a BPMN modeling solution that is consistent and integrated with the SCOrWare platform.

In year one, we have prototyped said solution by specifying BPMN-JWT mapping (Open Wide), developing (Obeo) the JWT to BPMN transformation, which uses model-to-model ATL technology and builds on the Eclipse STP BPMN editor using its flexible annotation framework, and providing a first integration of it in JWT.

Advised technology for new Eclipse graphical model editors is EMF / GMF. There is little point in developing another BPMN editor on EMF / GMF technology since the STP BPMN editor is already exactly that. That's why in year two, the stress is laid on cleaning the architecture of extensible JWT Transformations (jwt-transformation-base plugin, extension point and GUI) and enriching it, and developing the BPMN to JWT transformation. Note that ongoing work on by the University of Augsburg and Bull, supported by Open Wide, will also allow BPMN-like display of processes in additional GUI “views” in the JWT Workflow Editor.

and by developing tools allowing its edited diagrams to be transformed in SCOrWare formats (see other parts of this document)

BPMN integration with technical processes

A study comparing JWT and this BPMN's meta models and explaining how to match one with the other has been done jointly by Open Wide, Obeo and the JWT community.

Since BPMN shares with JWT the transition-based process paradigm, such a transformation is quite straightforward once the meaning of BPMN shapes in JWT has been decided and the information required by JWT added as annotations.

Obeo has provided a transformation from the JWT native meta model to this BPMN meta model (BPMN export) in year one on the basis on the previously mentioned study and Open Wide has integrated it as a graphical tool. In year two, Obeo plans to provide the opposite one (BPMN import in order to implement it technically using JWT).

See the “model transformation” part of the present document.

Note : the JWT meta model that will be the transformation target will be as following (Figure 27) :

A study comparing the various solutions for Business Process modeling and its relation to the SPEM modeling language from OMG (used for Development Process modeling) will be conducted by IRIT. The main purpose is to define how BP could be used to implement distributed process driven CASE tools. The study will explore the syntactic and semantic differences between the various languages, define model transformations and weaving allowing to relate them, and apply them on the case of some real development process defined in the TOPCASED project relying on the Eclipse Process Framework.

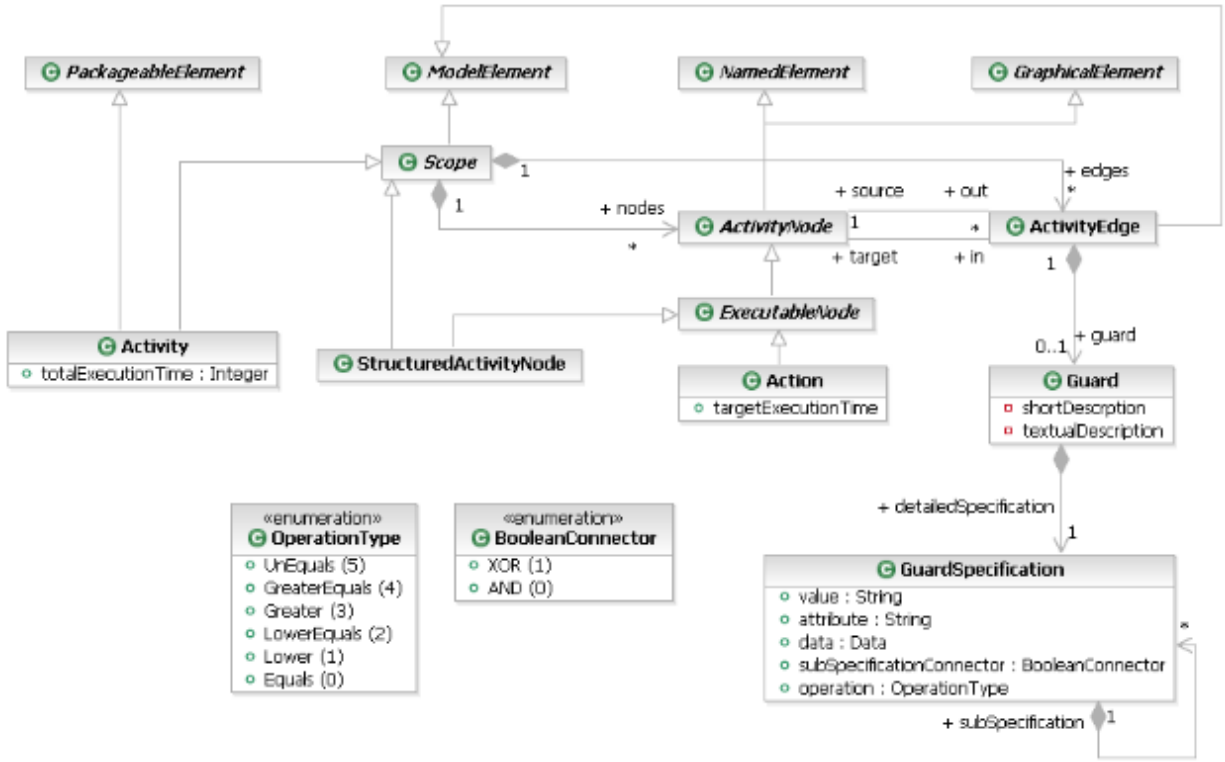


Figure 27: JWT Meta Model

4.3.3 Implementation

Study for establishing a BPMN subset consistent with the requirement of SCOrWare

This study has been done by Open Wide and using various additional sources (see part about BPMN to JWT transformation and Annex) to specify the SCOrWare requirements in terms of a business oriented representation of business processes.

This includes validating that this subset, along with annotations, will be enough to support all features of the SCOrWare BP solution.

The output is a document detailing the validation, along with unit tests as well as business tests consisting of commented sets of BPMN diagrams.

Extending Eclipse STP BPMN to support this subset

The Eclipse STP BPMN editor provides a simple, fully flexible annotation framework. This framework allows to enrich the BPMN diagram (and therefore its XML / EMF serialization) with additional information beyond what BPMN specifies, in the manner of collections of properties and their values. This additional information may typically be "implementation hints" allowing to map the BPMN diagram to a concrete implementation, namely to a "concrete", technical BP language like XPDL or BPEL, and to a concrete execution platform, i.e. BP engine or more generally in our context the SCOrWare platform.

We will extend the Eclipse STP BPMN editor this way to support said subset and every information required for transforming it to a format specific to the SCOrWare platform.

Those annotations will be for example :

on Task Activity,

- actionImplementation
- actionParameters
- actionValues

BPMN to JWT Transformation

See the other parts of this document.

Note : this transformation will be developed using Acceleo.

Since BPMN share with JWT the transition-based process paradigm, such a transformation is quite straightforward once the meaning of BPMN shapes in JWT has been decided and the information required by JWT added as annotations.

This transformation will be included as an Eclipse plug-in in order to be available to the user along the BPMN editor. We will develop this integration as a JWT Transformation and provide it in JWT.

Generic JWT transformation architecture

Work on such transformations shows that they are useful and have a wide range of use cases. They can be used to achieve bijective compatibility with similarly capable metamodels, to kickstart technical BP design from (or reverse engineer to) a business representation, or to export such technical model to a runtime executable format. They can be used to convert or even synchronize only parts of models, like an SOA PB's underlaying services, and going even further they can even be used to synchronize view specific information (like graphical position) across different views (like graph and swimlane), as JWT partners do.

To enable such benefits, a simple generic architecture has been defined and implemented for JWT Transformations. Most of it can be also used outside of JWT and at runtime. Its API, implementation and UI architecture can be found at http://wiki.eclipse.org/images/3/31/JWT_Transfo_archi.jpg .

Transformations for other BP and SOA design formats

Open Wide, EBM Web Sourcing and Obeo have collaborated to provide integration with SCA format. Open Wide has developed a useful feature allowing to create SCA-calling applications to be designed by mere drag-and-drop of an SCA composite in the JWT editor.

The University of Augsburg has developed (outside SCOrWare) a transformation of JWT to STP IM using model-to-model ATL technology.

These previous tools are contributed to JWT. Note that these tools can be used whatever the runtime technology used.

4.4 JWT technical designer

4.4.1 Objectives

The objective is to develop the tooling required by the BP developer use case of the top-down approach that BP brings to SOA.

This will be achieved with the help of JWT's format interoperability (JWT Transformations) and meta model genericity (JWT metamodel extensibility and Conf framework, which provides EMF Aspects and Profiles). Its relation to runtime (see next parts) must also be considered, so BP features that this editor allows to design can be executed on middleware implementing JWT Runtime APIs, and first and foremost Scarbo.

4.4.2 Specification

Choosing a standard executable process format and process engine to target

Obviously if we want business processes to run within the SCOrWare platform, a business process engine will be required that is able to orchestrate SCA / SCOrWare services (service call, data binding, even having a BP engine API made available to the SCOrWare platform).

As for now, JWT provides a simulator that supports its own BP format. This simulator is based on the jBoss jBPM technology.

A study has been done by Open Wide to choose the BP engine of choice for JWT / SCOrWare, that will support the SCOrWare use cases and the SCOrWare SCA and BP feature set.

Out of all BP engines Open Wide has experience with, Bull's OW2 Bonita has several items going for him. It is a fellow OW2 project, it has been integrated with the Petals ESB (by Open Wide also) in the context of the JoNES project, its next generation "Nova Bonita 4" will be built using the Process Virtual Machine core that promises to unify BP development and deployment - and actually will is the basis of the next generation of jBoss jBPM as well. It has all features expected of a modern workflow engine, including a web console, light and hot deployment, standard support (with XPDL). Finally, in November 2008, Bull and jBoss met with Open Wide and other JWT partners and decided that their next generation process editors will be based on JWT.

That's why Open Wide has chosen to develop support for the XPDL executable process format, and validate its tools at runtime by integration in OW2 Bonita. This support has been prototyped in year one, and after successful evaluation of it and of Nova Bonita 4, ported to Nova Bonita 4 within the larger context of the TaskEngineFramework in year two.

XPDL Capable designer

The aim of the technical designer is to provide a transition-oriented ("workflow") BP graphical designer compatible with the JWT tool suite.

As said above, the target chosen BP format will be XPDL. Compared to the native JWT BP format, it has the advantage of being standardized and as such is an open door to let SCOrWare processes be executed on the vast amount of compatible BP engines that are available.

This will be done by Open Wide, with the help of the JWT partners and community.

In year one, Open Wide has prototyped XPDL support within JWT by developing a transformation from the JWT meta model to the XPDL format.

In year two, the specification of the XPDL compatibility has been updated thanks to the inputs provided by the prototype, and adapted in the larger context of the TaskEngineFramework and the new Nova Bonita 4 engine. It has to be put in relation to Bull's latest involvment and code contributions to JWT. Final specification can be found at http://wiki.eclipse.org/JWT2XPDL .

BPEL and other BP executable formats

A BPEL designer is a big endeavor on itself. Moreover, it implies a range of issues that have not been definitely addressed for now, like choosing a preferred BPEL execution platform for SCOrWare, defining the level of SCA / BPEL specification supported by it etc. All the more since the use of BPEL to orchestrate services is something very common compared to the use of services within workflows, we have chosen to rather focus on specifying and integrating service orchestration in workflows. Finally, BPEL is a block-based scripting language, and therefore complex and conceptually awkward to transform to from "classic" oriented graph-based workflows.

However, it is to be noted that transforming JWT to BPEL is possible thanks to the University of Augsburg's CodeGenerator framework (available on Sourceforge's AgilPro). It has been used by Bull in an InfoQ article to support their affirmation that transforming BPMN to BPEL is not a good technical architecture for a BPM solution, which incidentally reinforces our above decision.

Open Wide has also written as an example a transformation from JWT to jPDL (the process format of JBoss' jBPM engine) using XSL technology. It has to be put in relation to jBoss' latest involvement in JWT.

4.4.3 Implementation

JWT to XPDL transformation

See the other parts of this document.

In year one, this transformation has been developed by Open Wide as a JWT Transformation in its own jwt-transformation-xpdl plugin using XSL transformation technology.

Developing a new XPDL editor would neither answer the problem of “design to implementation” views, nor address the numerous custom features that are specific to any given XPDL engine, nor even SCOrWare specific features. Therefore, in year two, the prototype developed in year one and producing XPDL for Bonita 3 has rather been cleaned and updated to Nova Bonita 4, but also has also been made more generic by adding the support of JWT Conf metamodel extensions and providing TaskEngineFramework helpers (see next parts).

This tool is contributed to JWT and Scarbo. Note that it can be used whatever the runtime platform to produce XPDL-compliant process definitions ; however, SOA features will only be available in process engines that implement JWT Runtime APIs, such as Scarbo's TaskEngineFramework implementation.

IRIT

A study comparing the most adequate BP description languages (BPEL, XPDL, JWT, ...) and execution engines for the project will be conducted by IRIT in relation with the appropriate parties. This study will be supported by prototype transformations between the core concepts of the languages. The key point of the study will concern the implementation in the other languages of the concepts that are available in some languages and not in the others.

5 Test and deployment tools

This Chapter concerns task 2.4. Section 5.1 specifies a set of tools to deploy SCA applications into PEtALS – FraSCAti, Section 5.2 a set of tools to test SCA applications from Eclipse, and Section 5.3 presents the graphical deployment designer allowing generation of deployment files in FDF or Tunes format.

5.1 Deploying SCA applications into PEtALS

When the SCOrWare project ends up, PEtALS will propose a service engine based on FraSCAti to run SCA composites. For the SCA aspect, PEtALS will require the same artifacts than FraSCAti. However, since PEtALS is a distributed enterprise bus based on the JBI standard, the deployment step differs from other runtime platforms. More precisely, the SCA artifacts must be packaged with a JBI descriptor inside a service assembly archive.

The terms “JBI descriptor” and “service assembly” are specific to the JBI specification.

To give a better explanation of this deployment step, let's take an example.

As it was said before, PEtALS is a distributed ESB. Let's assume we have a network with two PEtALS nodes A and B (a node is a machine with a running PEtALS platform). Let's also suppose that on node A was deployed the SCA service engine (the PEtALS component based on FraSCAti to run SCA composites).

A user can now deploy an SCA composite on any node, A or B, provided that this composite is packaged as a service assembly. A service assembly is a zip archive with the following structure:

- META-INF
  - jbi.xml
- service-unit.zip

The content of the jbi.xml file for an SCA service assembly looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi:jbi version="1.0"
         xmlns="http://java.sun.com/xml/ns/jbi"
         xmlns:jbi="http://java.sun.com/xml/ns/jbi">

    <jbi:service-assembly>
        <jbi:identification>
            <jbi:name>sa-SCA-Calculator-provide</jbi:name>
            <jbi:description>The deployment artifacts...</jbi:description>
        </jbi:identification>

        <jbi:service-unit>
            <jbi:identification>
                <jbi:name>su-SCA-Calculator-provide</jbi:name>
                <jbi:description>
                    The service name of the SU embedded into this zip file.
                </jbi:description>
            </jbi:identification>

            <jbi:target>
                <jbi:artifacts-zip>su-SCA-Calculator-provide.zip</jbi:artifacts-zip>
                <jbi:component-name>petals-se-sca</jbi:component-name>
        </jbi:target>
        </jbi:service-unit>
    </jbi:service-assembly>
</jbi:jbi>
```

The zip file contained in the service assembly is a service unit (the name of the service unit zip file can be different from “service-unit”). It is in this archive we package the SCA composite.

A service unit is a zip archive. With SCA, it has the following structure:

- META-INF
  - jbi.xml
- composite files
- WSDL files
- implementation sources (Java only)

The content of the jbi.xml file for an SCA service unit looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- JBI descriptor for the PEtALS component "SCA", version 0.1-Snapshot -->
<jbi:jbi version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jbi="http://java.sun.com/xml/ns/jbi"
    xmlns:petalsCDK="http://petals.ow2.org/components/extensions/version-4.0"
    xmlns:sca="http://petals.ow2.org/components/sca/version-1">

    <!-- Import a Service into PEtALS or Expose a PEtALS Service => use a BC. -->
    <jbi:services binding-component="false">

    <!-- Import a Service into PEtALS => provides a Service. -->
        <jbi:provides interface-name="ICalculator" service-name="Calculator"
                      endpoint-name="CalculatorEndpoint">

            <!-- CDK specific fields -->
            <!-- WSDL file (case: composite with one service) -->
            <petalsCDK:wsdl>calculator.wsdl</petalsCDK:wsdl>

            <!-- SCA specific fields -->
            <sca:source-jar>impl.jar</sca:source-jar>
            <sca:composite>calculator.composite</sca:composite>
        </jbi:provides>
    </jbi:services>
</jbi:jbi>
```

In fact, the service assembly tells PEtALS which component a service unit must use. In this case, the service assembly tells PEtALS the included service unit works with the SCA component. The content of the service unit is then used by the SCA component to deploy and run the top-composite.

As you can notice, it is a fastidious task to package an SCA composite to run it into PEtALS. In general, people use Maven to generate their service assemblies. However, this is not as practical as it should be. This is why tools are proposed to speed up and facilitate this task.

5.1.1 Generating JBI packages in the generic way

This first proposed tool is said “generic” because it can be used to create deployment and configuration archive for every PEtALS component (SCA-FraSCAti, BPEL-Orchestra, XSLT, SOAP...).

The way it works

It consists in a set of Eclipse plug-ins with wizards asking for fields to be filled by the user and which then generate a project containing:

- The jbi.xml file for the service unit.
- The files / resources whose location were given in the wizard.

Once this project has been generated, the user can edit the jbi.xml file by hand if necessary (the wizard does not cover advanced aspects).

Eventually, a right click on the project followed by the action “Package for PEtALS” builds the service unit and the service assembly for PEtALS. The resulting archive is saved into the generated project.

Target users and developers

The users of the tooling itself are expected to be PEtALS users, which manipulate one or several components of PEtALS. The SCA-FraSCAti service engine might be one of them. This tool aims at facilitating the import of services into PEtALS, the exposition of PEtALS services outside the bus, and the configuration of service engines.

Besides, this tool must respect some constraints for its development, packaging and maintenance:

- Since PEtALS users do not use mandatory all the components, this tool should be able to embed only the required parts for the components the user needs. As an example, if a user only installed the SCA and the SOAP components, he should be able to download only the tooling for the SOAP and the SCA components.
- Each part of the tooling associated with a component should be maintained by the person who developed the component for PEtALS. Since PEtALS developers do not all have knowledge about Eclipse and do not have time to learn more about it, the tooling should provide an easy and fast way to create and maintain the tooling for a component, with no specific knowledge of Eclipse required. This is an important request for this tool: the less possible Eclipse development.

Organization

The main task performed by this tool is the generation of the jbi.xml file for the service units. The content of this file varies from one component to another. However, they share a large part of their content.

More exactly, we can see three different kinds of data in the jbi.xml:

- The JBI part, inherited from the JBI specification (stable).
- The PEtALS specific part, related to the version of PEtALS.
- The component specific part, related to the component the service unit is intended.

The wizards should reflect this organization. Given this information and the previous requirements, it leads to the following organization (Figure 28).

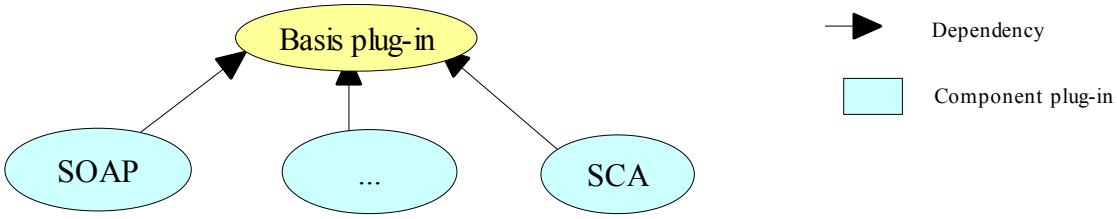


Figure 28: PEtALS plugin-ins organization

1. The basis plug-in is in charge of the common parts among the components. It can be seen as a framework which generates plug-ins at runtime from component plug-in resources.
2. Each of the component plug-ins is in charge of providing the specific part of the component. Since component versions and PEtALS versions are not always correlated, these plug-ins also provide the elements related to the PEtALS version. This way, each time a new version of PEtALS or of a component is released, the associated plug-in will be updated independently of the others. This also allows the users to select which tools they want. The only requirement to run a component plug-in is to have the basic plug-in. A component plug-in may support several versions of this component, each version being independent of the others.

This organization allows users to manage their plug-ins easily with an update site (Figure 29): choose which component plug-in to install, update their plug-ins.

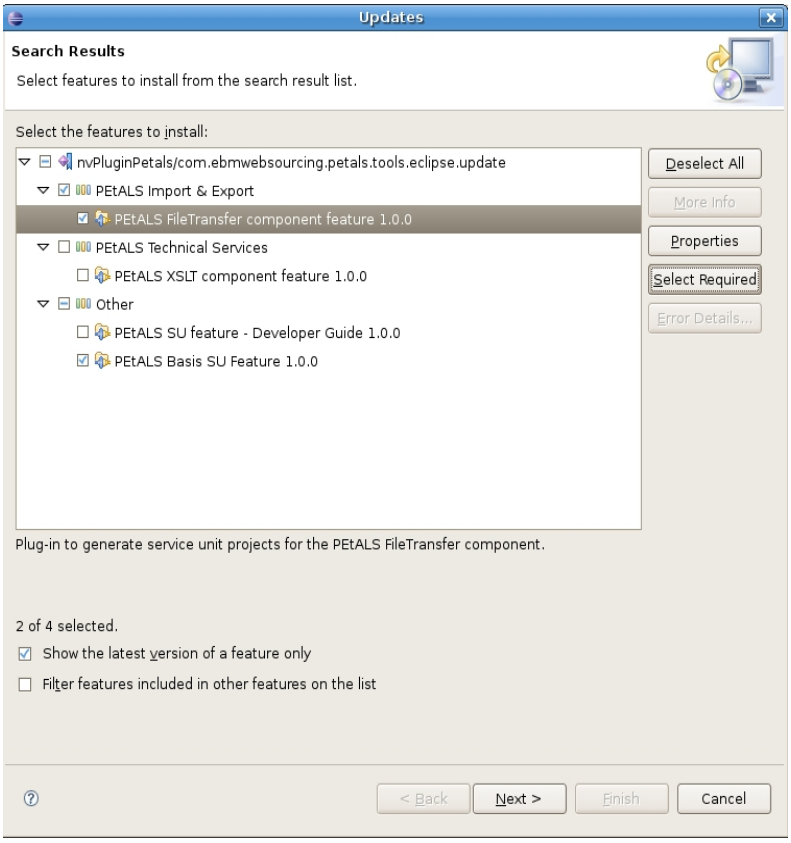


Figure 29: PEtALS installation

The wizards

There are three kinds of wizards:

- Import a service into PEtALS, which makes a service visible into PEtALS.
- Expose a PEtALS service, which makes a PEtALS service visible from outside the bus.
- Use a technical service, which configures a service engine provided by PEtALS.

Each wizard category proposes the components whose plug-ins have been installed. For the two first ones, the plug-ins are displayed as protocols (e.g. “Using SOAP”, “Using JMS”), while for the last one, the plug-ins are displayed as the name of the service main feature (e.g. “Using XSLT”, “using SCA”).

A wizard is made-up of at least two pages:

- The version page: since PEtALS components have versions and that the jbi.xml file they can accept depends on this version, it is asked to the user to choose the version of the component he is using. If only one version is available, this page is automatically skipped.
- The general page: this page asks for the basic information related to JBI (service, interface and endpoint names, WSDL location or referenced PEtALS service, name of the project the wizard has to create).
- The component and PEtALS version specific page(s). It is contributed directly or indirectly by the component plug-in.

When the wizard completes, it creates the project with the name provided in the wizard. It also imports the files whose location were given in the wizard (e.g. the WSDL file) in the created project. Eventually, it creates the jbi.xml file from the information given in the wizard (Figure 32).

Obviously, some help is provided in the wizards to improve the usability. In the general page, if the user provides a WSDL file (versions 1.1 and 2.0), the service, interface, endpoints and project names are extracted or generated from this file. Besides, each wizard page (Figure 30, Figure 31) proposes a help section giving more information about the page and the information it requires (what are they useful for?)

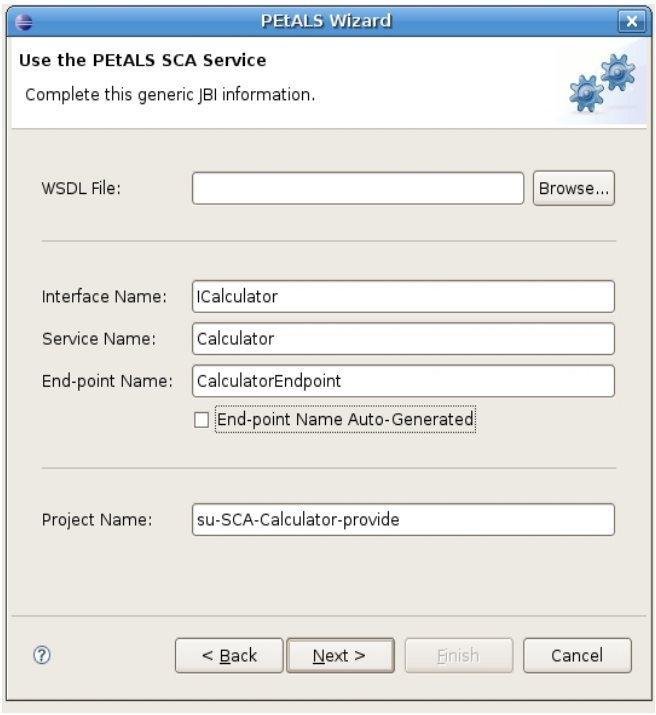


Figure 30: The "general page" for the SCA component

Give examples of values for some fields...). Eventually, as specified in the Eclipse User Interface Guidelines, every field is validated with respect to rules defined in the wizard, so that the user knows if he can go to the next page or why he can't in the case where the “next” and/or the “finish” buttons are disabled.

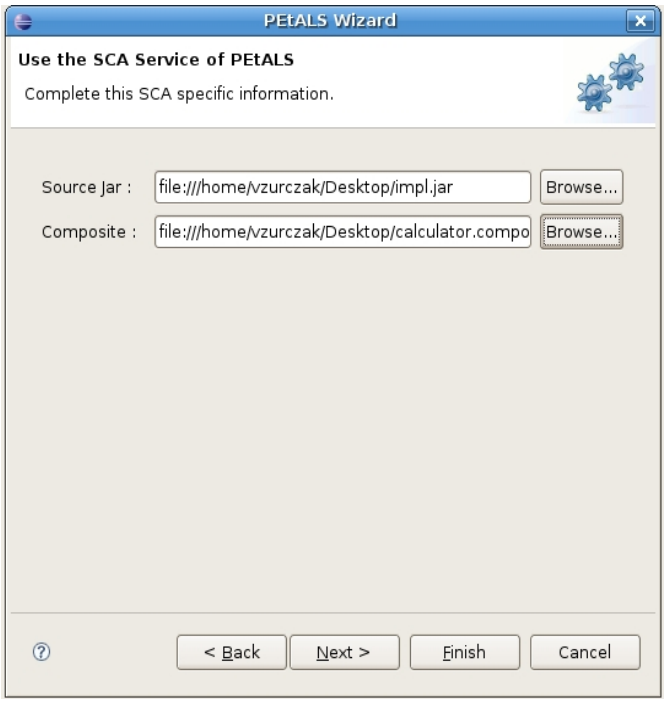


Figure 31: The wizard page asking for the SCA component specific fields

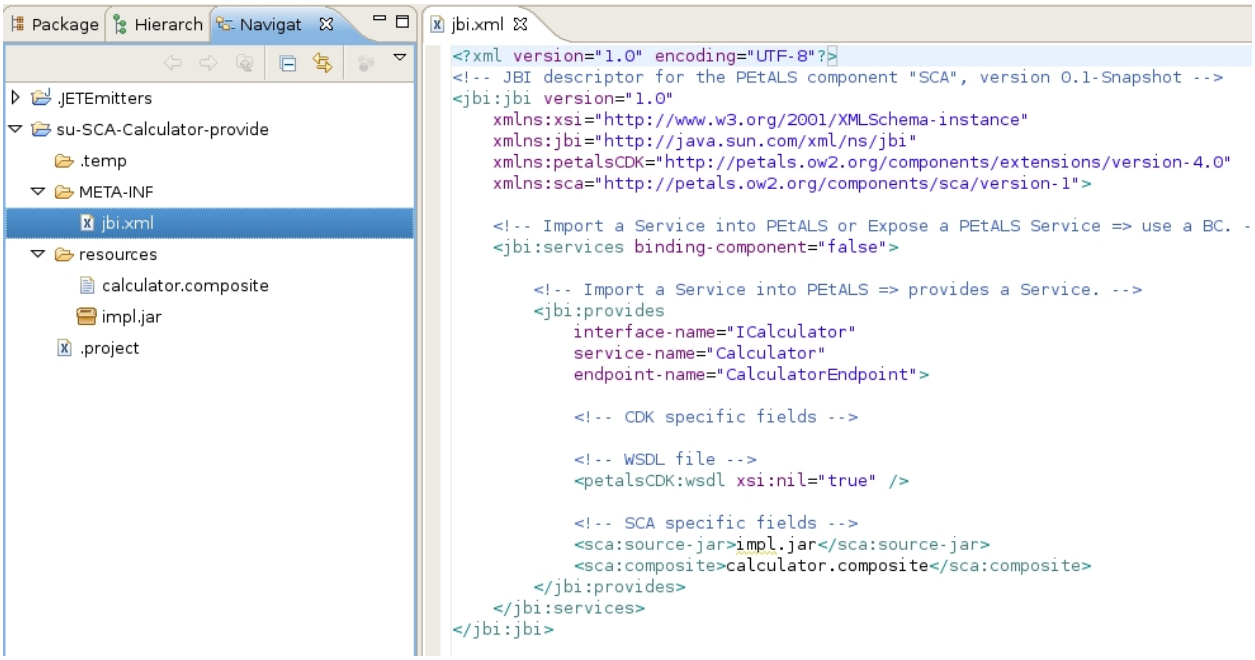


Figure 32: The artifacts generated by the wizard, a project ready to be packaged for PEtALS

Deployment archive

The generated project has the following structure:

- .temp
- META-INF
  - jbi.xml
- resources
  - < imported files >

This project itself can not be deployed into PEtALS.

By right-clicking on it and selecting the action “Package for PEtALS”, a service assembly is generated from this project and saved into this same project. The generated archive file is a valid deployment archive, ready to be deployed into PEtALS, either by using the administration console of PEtALS, or by using the PEtALS Eclipse view (see section 2.1.3).

When looking at the generated service assembly, its service unit contains the jbi.xml file and the imported resources which were in the project.

Plug-ins integration and coherence

The basic plug-in is intended to provide the common parts of the components. However, most of the information is brought by the component plug-in (versions, component specific data and information specific to a PEtALS version). Moreover, it was required that these component plug-ins would not embed Eclipse code.

As a result, and in coherence with the discussion within the PEtALS team, it was decided the component plug-ins would have the following structure:

- src
- bin

- icons
  - wizban
    - < icons for the wizard banners >
- resources
  - 2.0
    - < component's XSDs >
    - CDK
      - < PEtALS's XSDs – related to the version of PEtALS >
  - < other version of the component >
- < default Eclipse plug-ins artifacts (plugin.xml, META-INF...) >

The wizards have to be registered dynamically into the plat-form. The content of their pages has to be generated by the basic plug-in by using the "resources" folder of the component plug-in. Its sub-folders are used to define the content of the "Version page" of the wizard. The "General page" is fixed by the basic plug-in. Eventually, the following pages are generated by the basic plug-in from the XSD files provided by the component plug-in.

To make sure all of this works, the basic plug-in defines an extension-point component plug-ins have to use. By registering themselves to this extension-point, the component plug-ins will make themselves visible by the basic plug-in.

This fully respect the requirement "no specific knowledge of Eclipse is required". The XSD files used in the component plug-ins are the same than those used by PEtALS to validate jbi.xml files. Thus, we can reasonably think that a component plug-in may be developed in few minutes once its XSD files are provided. The evolution of the versions is managed by adding a new version sub-folder to the "resources" folder and by updating the plug-in version in the plugin.xml file. This way, maintaining this tooling should remain an easy task.

The main task here is the development of the basic plug-in.

The SCA component plug-in

The XSD files of the SCA component indicate only two fields:

- Composite: the top composite file.
- Implementations: the jar file containing the implementations of the composite.

Packaging

In a first time, this tool should remain in the EBM WebSourcing forge. It might then become a contribution to the OW2 community or in the Eclipse STP project. In any case, it is an open-source tool, licensed under EPL.

5.1.2 Generating JBI packages in an SCA specific way

SCA developers who would like to test their application within PEtALS, and who are not used to PEtALS or JBI, will not ever use the previous tooling to deploy their SCA application into PEtALS.

Besides, most of the information required in the "generic packaging generator" can be extracted from the top composite file (which makes the previous tool quite nice for PEtALS users in general, except for the SCA component). Hence, it is possible to generate a service assembly directly from an SCA project.

No need to create a project here, we reuse the existing SCA project. The service name is the name of the composite. The interface name is the name of the composite followed by the string "Interface". Eventually, the endpoint name is let to PEtALS to be generated (endpointname = "petals:autogenerate"). The fields required specifically by the SCA service engine are the name of the top composite and optionally a jar file containing the implementations. These two last fields can be retrieved automatically when selecting a composite file in the work space.

Usage seen through a scenario
Let's suppose a user has developed an SCA application in Eclipse with the SCA tooling of STP and that he now wants to deploy it into PEtALS.

1. The user creates a PEtALS server (see section **2.1.3**).
2. The user adds his project to this server (in the server property).
3. When he starts this project in the server, the tooling generates all the required artifacts and deploy them on PEtALS.

The service assembly generation process is straight-forward:

- Create the jbi.xml file(s) for service unit(s).
- Package the implementation into a jar file.
- Gather the generated jar file, the jar dependencies and the jbi.xml file(s) into service unit(s).
- Generate the jbi.xml file for the service assembly.
- Generate the service assembly zip file.

This process is performed in background, the user does not see it.

If an error occurs, an error is displayed in the console.

This tool does not check if the project embeds unsupported implementations, like BPEL or scripts. It is assumed the user knows which implementations are supported by PEtALS (only Java for instance). If the deployment succeeds, the console provides a report indicating that the deployment succeeded, in the same manner that what is displayed when you deploy something on a Tomcat server from Eclipse.

The user should then be able to monitor its composite in the PEtALS administration console .

This tool will be proposed to the Eclipse STP project to integrate the SCA sub-project as soon as it reaches a stable version. License: EPL.

5.1.3 Eclipse PEtALS server and server view

The last tool proposed within this section is an Eclipse view which allows a PEtALS developer to interact with a PEtALS server from Eclipse. A *PEtALS developer* is somebody who develops artifacts that can be deployed inside PEtALS. These artifacts can be either service units and services assemblies, or JBI components (service engines or binding components).

PEtALS server

A PEtALS server is defined as any other server in Eclipse, by using: **File > New > Other > Server > Servers**

and then by selecting: **OW2 > PEtALS** <*version*>

A wizard page is then displayed, asking for required information to define the server. Once all the mandatory fields are filled in (these fields may vary from one version to another), the user clicks OK. The result of this is the registration of a new PEtALS server, visible in the *servers view*.

PEtALS in the "Servers" view

This "Servers" view can be started by selecting **Window > Show View > Server > Servers**.

This view displays all the project associated with this server.

Interactions between PEtALS and Eclipse are made using JMX, in a local mode (Eclipse and the PEtALS server are on the same machine). This view is only intended to developers, not to monitor or administrate a PEtALS server (for these tasks, there is the administration console of PEtALS).

Interactions with PEtALS

Actions:

Actions are operations available in a context menu after a right click on an element.

The available actions on a PEtALS server are the following:

- Start: available only if the artifact is not already started. Starts the artifact.
- Stop: available only if the artifact is not already stopped. Stops the artifact.
- Shutdown: available only if the artifact is not already shutdown. Shutdown the artifact.
- Show Properties: displays the properties of the artifact in the properties view.

Actions should also be registered into the SCA common plug-in of Eclipse STP to run a *.composite file on PEtALS (the action triggers the deployment of the project into PEtALS using the mechanisms provided by this view). Same thing should be added for jbi.xml files inside the generic JBI packaging generator basic plug-in.

Preferences:

A preference page is available to specify extra-information about PEtALS configuration.

In this page, the user can define a directory containing JBI components. These components will be used in case where the user deploys service units which use JBI components not deployed into the registered PEtALS server.

Once the user has chosen a directory on its file system, it is introspected to determine which components are present in it. The list of present components is then displayed into the page, under the browsing widget.

For example, let's suppose a user has created an SCA project and that he wants to test it in PEtALS. In this scenario, we also suppose he has not deployed the SCA service engine (based on FraSCAti) into PEtALS.

Now, if this user has set the “Components” directory into the PEtALS server preference page, and that this directory contains the SCA JBI component, then deploying the SCA project on the PEtALS node in the server view results in the deployment of the SCA service engine followed by the deployment of the SCA project into PEtALS.

Packaging

This view will be proposed into Eclipse STP as soon as it reaches a stable version. License: EPL.

5.2 SCA applications test

One important aspect that can influence a team or a company to choose SCA for its development is the efficiency and the cost of this development. Clearly, SCA provides a powerful and very easy way to write distributed applications. But if these applications are badly tested or that tests are painful to write and perform, there are few chances for SCA to be adopted.

This is why tools should be proposed to ensure SCA can be used for industrial projects, where tests lead developments and have a major importance.

However, SCA is a brand-new technology and make tools for testing is a little soon.

The first goal of this part is the writing of a document explaining the possible leads to make such tools. It will explore the usual tools and list the available leads.

A secondary but optional objective is the development of testing tools.
The approach they would adopt is similar to what another open-source SCA runtime plat-form already did (it is neither FraSCAti nor Tuscany, it is called Fabric3). This project has defined a way to run unit and integration tests for composites using Maven. This is achieved by using JUnit and Easymock, two open-source products. Fabric3 has defined implementations to run and use these tools (through what they call “junit” and “mock” implementations).

What we might do here is to reuse the concepts and the tools adopted by Fabric3, integrate them into Eclipse and make them work with PEtALS–FraSCAti (and ideally with any platform supported by the Eclipse SCA project).

5.2.1 Unit testing for components and composites

Unit testing for SCA is about testing services of a component or a composite and ensure their contract is fully respected. For any element to test, the user has to write TestCases (as he would do for any JUnit test) where he instantiates the services, mock the references, executes operations and checks assertions on the results.

No tool can be provided to say whether the user wrote a good test or not. However, we can:

- Bring support and usability when defining a test and managing testing code (e.g. which component / composite is this test for ?).
- Improve the speed of development, e.g. by setting JUnit and Easymock directly in the classpath and provide some templates of TestCase.
- Allow users to run JUnit tests from Eclipse and display the results in the usual way Eclipse displayed JUnit test results.

This last step requires to generate a composite that embeds the TestCases and the component / composite to test. Here are two patterns of unit test (Figure 33, Figure 34).

Test a composite

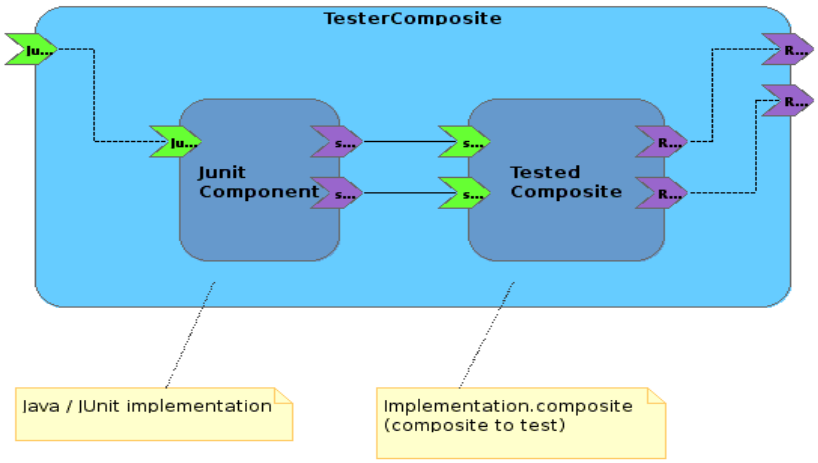


Figure 33: Pattern used to test a composite

Test a component

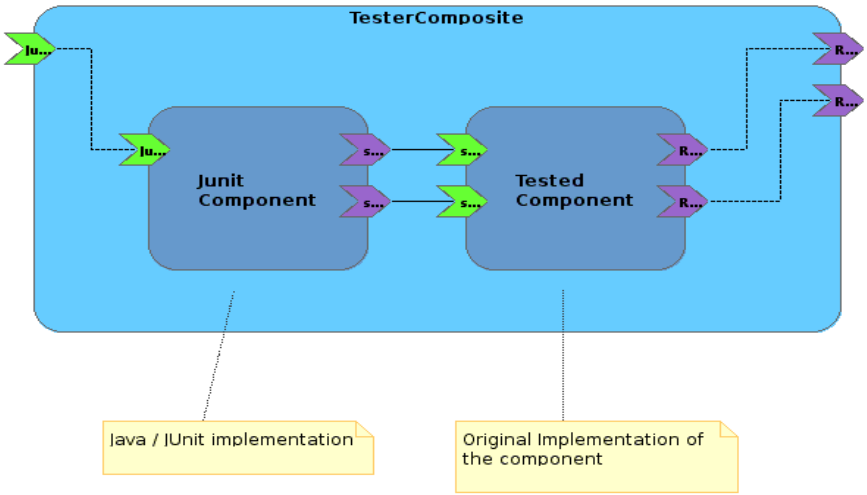


Figure 34: Pattern used to test a component

In these patterns, the references of the tester composite are all mock objects, whose behaviour is specified in the TestCases (either in testing methods or in setUp / tearDown methods) and eventually in the composite (by replacing orginal references by references to mock objects). These ock objects are built using EasyMock. It is also important to notify that the tested element services are all linked to the references of the JUnitComponent. These references are used in the TestCases.

N.B.: it is possible to define other patterns to test any other service. This could be the basis in the Eclipse STP project to build a test platform for services, whatever their bindings are.

Here is a sample of a TestCase used in Fabric3...

```
package org.example;

import org.osoa.sca.annotations.Reference;
import junit.framework.TestCase;

/**
 * Tests the retention of composite file order for injected components
 */
public class OrderingITest extends TestCase {
    private ItemDisplayService displayService;

    @Reference(name = "displayService")
    public void setDisplayService(ItemDisplayService displayService) {
        this.displayService = displayService;
    }

    public void testOrderedInjection() {
        String[] expectedItemNames = {"ONE", "THREE", "TWO"};
        Item[] actualItems = displayService.getItems();
        assertEquals(expectedItemNames.length, actualItems.length);

        for(int idx = 0; idx < expectedItemNames.length; idx++) {
            assertEquals(expectedItemNames[idx],actualItems[idx].getName());
        }
    }
}
```

... with the composite file to test...

```
<?xml version="1.0" encoding="UTF-8"?>
<composite
          xmlns="http://www.osoa.org/xmlns/sca/1.0"
          xmlns:fabric3="http://fabric3.org/xmlns/sca/2.0-alpha"
          name="ExampleTest" autowire="false">

    <service name="OrderedDisplayService"
            promote="OrderedDisplayService/ItemDisplayService" />

    <component name="OrderedDisplayService">
        <implementation.java class="org.example.OrderedDisplayService" />
        <reference name="items" target="Item1 Item3 Item2" />
    </component>

    <component name="Item1">
        <implementation.java class="org.example.ItemImpl" />
        <property name="typeName">ONE</property>
    </component>
```

```
    <component name="Item2">
        <implementation.java class="org.example.ItemImpl" />
        <property name="typeName">TWO</property>
    </component>

    <component name="Item3">
        <implementation.java class="org.example.ItemImpl" />
        <property name="typeName">THREE</property>
    </component>
</composite>
```

... and the tester composite...

```
<?xml version="1.0" encoding="UTF-8"?>
<composite
          xmlns="http://www.osoa.org/xmlns/sca/1.0"
          xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
          xmlns:f3="http://fabric3.org/xmlns/sca/2.0-alpha"
          name="OrderingComposite" autowire="false">

    <component name="OrderedInjectionTest">
        <f3:junit class="org.example.OrderingITest" />
        <reference name="displayService" target="OrderedDisplayService" />
    </component>

    <component name="OrderedDisplayService">
        <implementation.composite name="OrderedDisplayService"
                    scdlResource="org/example/ordered-service.composite" />
    </component>
</composite>
```

As you can see, the JUnitComponent has here a junit mark-up (implementation) which refers to a specific module of Fabric3. What we should do (or try to) is to find a solution to run TestCases in a Java implementation, so that any SCA runtime can run it.

Notice also that the use of EasyMock is not fully described here, since there are still work to do on how we can integrate it in a general approach. The basis idea is to reuse what Fabric3 proposed and to generalize it.

Notice that the TesterComposites can be generated automatically from the composites to test. The remaining big issue about this is the management of TestCases in the work space. Most likely, it seems that generating a testing project (with Java and SCA natures) is the best way to separate the code to test from the testing code. Managing TestCases in the test project then relies on an organization and conventions in the resource system (e.g. name of the folders).

A user would be able to run a TestCase by right-clicking on it and selecting "Run As > Sca Junit > <platform>" (if a platform was registered in the environment). The tool would then generate the required composite files, package and deploy them to the platform, generate and run a client to start the tests, get the results and display them to the user.

5.2.2 Integration testing within SCA applications

In the previous section, we have explained that in unit testing, the tested elements should be isolated from the external world by mocking their dependencies (references). Integration test is a bit more harder to treat since there are many ways to perform them.

One solution, proposed here, consists in running TestCases where mock objects are progressively replaced by the real elements (components when references are wired with a component service, or services when references are promoted by a composite reference).

What has to be done here is to find a good organization of the tests in the workspace and in the projects. This is what has been said in the previous section, but if we want to run integration test, this part must be clearly reinforced. To improve usability, one track could be to reuse the SCA Composite Designer, so that you can graphically define which elements are mocked and which one are not (e.g. by greying mocked elements).

There is a lot of work to do here, just to design an efficient way of managing these tests.

5.2.3 Running and Debugging tests from Eclipse

As it was said before, TestCases can be run to test a component or composite. These TestCases should be able to be run on any platform supported by Eclipse, provided that the user has registered a server for the selected platform.

One additional feature that might help developers is the debugging. Users should be able to run TestCases in debug mode, that is to say they can set breakpoints into the Java implementations of components. When a breakpoint is reached during the execution, the user is asked whether he would like to launch the debug perspective to control the breakpoint. This the same mechanism used when debugging Java programs.

For instance, it is not known whether platforms need some changes to support debugging.

One possible extension of this (not expected to be done during this project) is the creation of a "Service Explorer", in the same way than (or by extending) the "Web Service Explorer" created by the WTP project for Web Services. This "Service Explorer" would allow to call any service of a component / composite by filling in a web form and getting the result on screen. Debug support would be here useful during development.

Every tool related to test in section **2.2** will be proposed to the Eclipse STP project to integrate the SCA sub-project as soon as it reaches a stable version. License: EPL.

5.3 Deployment designer

5.3.1 Objectives

This tool provides a quick and easy way to construct graphically a “generic” deployment plan of SCA applications. The deployment plans are generic in the meaning that they are independent of any deployment tool. From these plans, this tool generates deployment plans specific to some deployment tools like FDF or Tune (IRIT new version of Jade).

This tool allows :

- To create new generic deployment plans.
- To modify existing generic deployment plans.
- To generate deployment plans for FDF and Tune.
- To import FDF or Tune deployment plans.

5.3.2 Specification

First, we define a meta model for the deployment of software archives, SCA runtime archives and SCA applications on a set of machines. This is the meta model for the generic deployment plans. Second, we propose a graphical representation for the elements of this meta model and functional specifications for the graphical deployment designer.

Deployment meta model

Deployment plans must respect the following meta model (Figure 36 represents graphically this meta model):

- **DocumentRoot**: the root of a deployment plan is composed of
  - One reference to the local repository (LocalRepository) that contains softwares (e.g. JDK) and SCA runtime archives that can be deployed.

  - 1..n references to the machines where SCA applications must be installed (SCAHost).
  - 1..n SCA archives containing one or more SCA composite description and their implementation (SCAPackage).
- **LocalRepository**: the local repository contains
  - 0..n software archives (SoftwareArchive).
  - 1..n SCA runtime archives (RuntimeArchive).
- **SoftwareArchive** (a software archive) and **RuntimeArchive (**an SCA runtime archive) are described by the path and the name of the archive.
- **SCAHost**: describes a machine with information allowing to access it and upload softwares, SCA runtime, and SCA applications. It is composed of
  - The name of the host, the type of transfer, protocol and shell to use.
  - One user (**User**). A user is described by its user name, its password and/or its key.
  - 1..n reference to the SCA runtimes (SCARuntime) that are installed on the host.
  - 1..n reference to the software archives that are installed on the host. For each one the directory where to install it (Software).
- **SCARuntime** (**Software**) represents a particular SCA runtime archive (software archive) installed on a particular host.
- **SCAPackage**: represents an archive containing one or more SCA composite description and their implementation. It is composed of
  - The path and the name of the archive.
  - A reference to the SCA runtime.
  - 1..n references to the Composite that form part of this archive.
- **Composite**: this element extends Composite defined in the SCA specification.
  - composite: the name of the composite file.
  - 0..n references to other Composite. These references represent business dependencies between SCA composites.

Graphical deployment designer

This tool allows to describe the deployment of software archives, SCA runtime archives and SCA applications on a set of machines. Elements of the deployment meta model defined above are represented on follows:

- A local repository is represented by a rectangle. This figure is divided in compartments where each compartment contains a JDK archive or an SCA runtime archive.
- A machine is represented by the figure defined in Figure 35. The name of the machine is on the top of the figure.

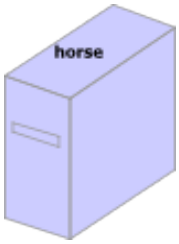


Figure 35: Graphical representation of a machine

- User of a machine is represented by the following figure:  . This figure is on the bottom left of a machine.
- Deployment of SCA runtime are represented by an orange rectangle on the machine representation.
- The Composite is represented in the same way as a Composite in the Composite Graphical Designer (Section 4.1). This designer allows to redefine Service, Reference, and Property of a Composite. A Wire between SCA applications is represented by a black line between service and reference.
- Business deployment dependencies between SCA applications are represented by a red line and a label to show the dependency type.

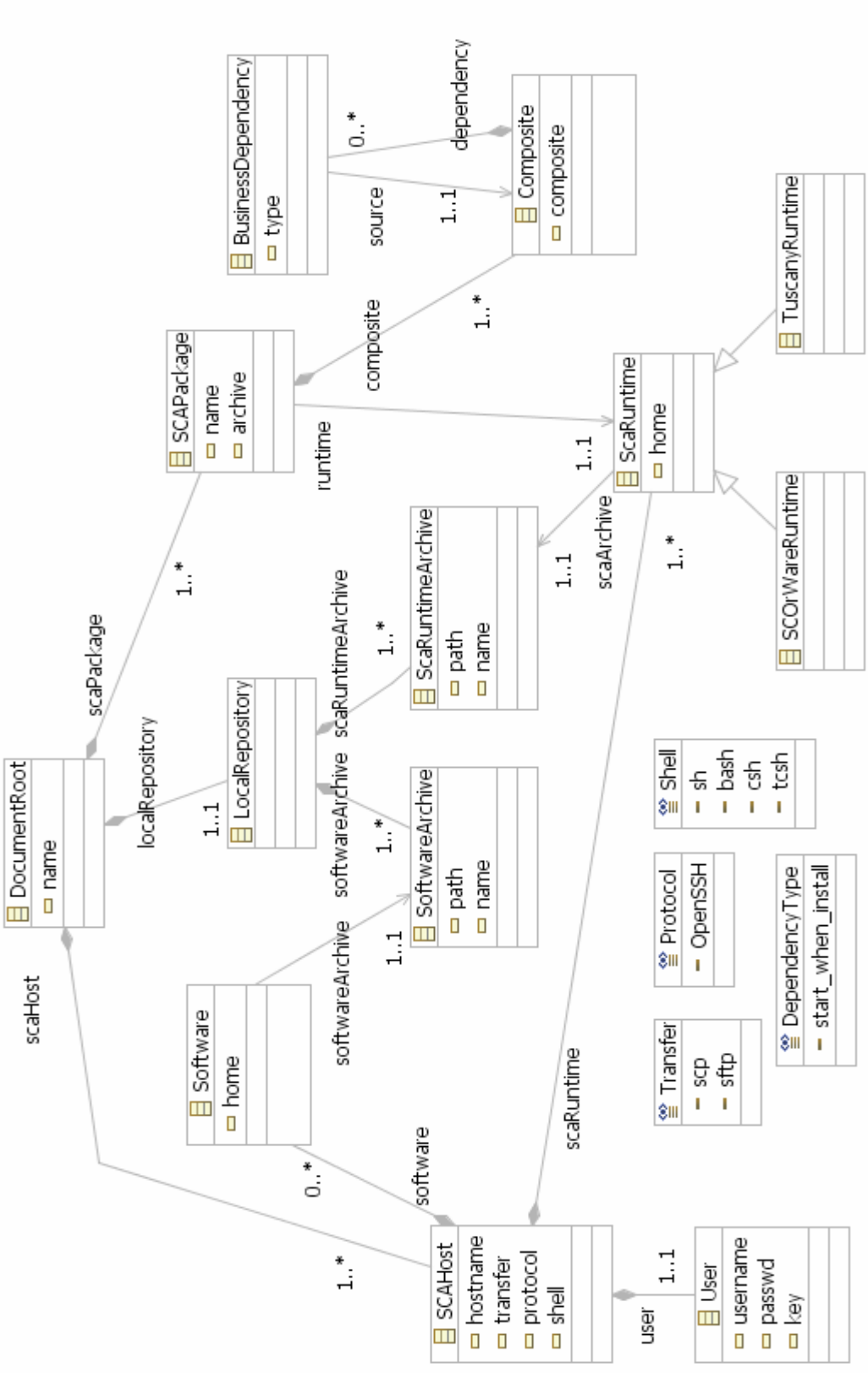


Figure 36: Deployment meta model

5.3.3 Implementation

The deployment graphical designer will be implemented with the GMF framework as shown in Section 4.1.3. The generation of FDF and Tune configuration files will be implemented with Acceleo.

FDF Mapping

The first part of an FDF file defines the system architecture with information allowing to access the machines and upload software. For each host, FDF needs the host name, the user, the transfer method, the protocol, the shell and the list of softwares to install. These informations are in the meta model elements: SCAHost, User, and SoftwareArchive.

The second part of an FDF file contains a description of the SCA runtime. For each host, one or more SCA runtime can be described. For each SCA runtime description, FDF needs the SCA runtime archive path and name, and the home directory where the SCA archive will be installed. These information are in SCAHost, SACRuntimeArchive and SCARuntime meta model elements.

The third part contains the description of the SCA applications.

The figure 37 shows an example of an Acceleo template that generates the description of the hosts in the FDF format.

```
1  # Description of hosts
2  Hosts = INTERNET.NETWORK {
3   <%for (scaSystem){%>
4    <%hostname%> = INTERNET.HOST {
5      hostname = INTERNET.HOSTNAME(<%hostname%>);
6      user     = INTERNET.USER(<%user.username%>,<%user.passwd%>,<%user.key%>);
7      transfer = TRANSFER.<%transfer%>;
8      protocol = PROTOCOL.<%protocol%>;
9      shell    = SHELL.<%shell%>
10     software {
11         java    = JAVA.JRE {
12         archive = JAVA.ARCHIVE(<%java.javaArchive.path%>
                                   <%java.javaArchive.name%>);
13         home    = JAVA.HOME(<%java.installDir%>);
14       }
15     }
16   }
17  <%}%>
18 }
```

Figure 37: Example of Acceleo template for FDF

Tune Mapping

One of the purpose of the SCOrWare project is to establish bridges between the various technologies involved in the project. Tune will therefore rely on some services provided by FDF in order to implement autonomic administration. Currently, Tune relies on UML models in order to express the system architecture and to configure the autonomic components. SCOrWare will allow to improve this solution by relying on dedicated tools both for expressing and modeling the system architecture and deployment.

The DSL and graphical designer will be implemented by Obeo with the help of INRIA (FDF tools) and IRIT (Tune tools).

Figure 38 shows a prototype of the deployment designer developed by Obeo.

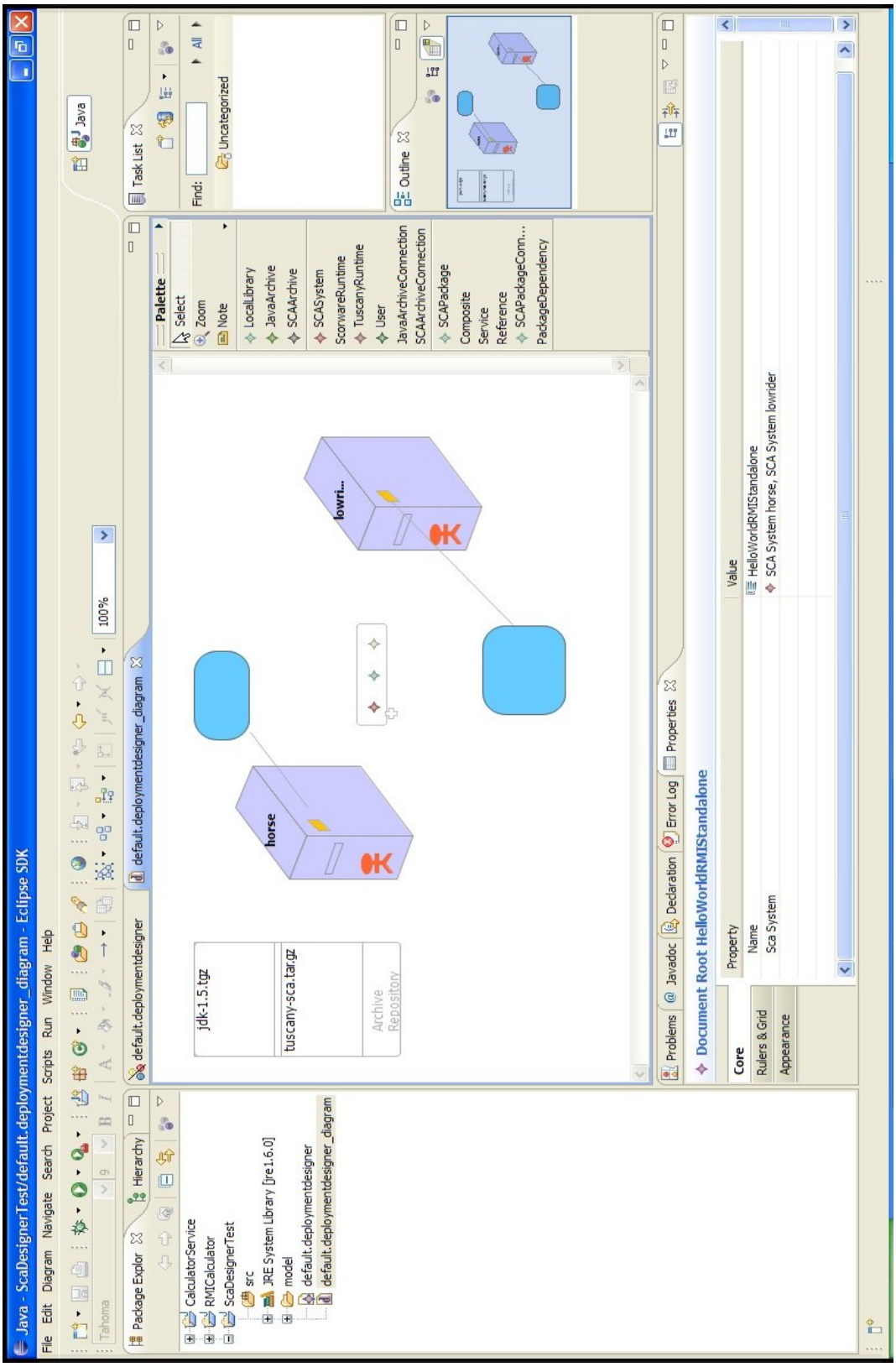


Figure 38: Deployment designer prototype

6 Tools for searching and semantic composition of services

The objective of the task 2.5 is to help the developer and integrators building on the SCOrWare platform to find and better reuse SCA services / components in a consistent manner.

This is achieved by enriching tooling, notably the JWT Designer to the business process developer and the SCA Composite Designer, by features allowing to design (SCA) service composition and use semantics to improve such service composition's reusability and consistency.

Semantic search relies on an SCA / SCOrWare service / component registry sitting on top of INT's Trader, in similarity to INT's work in WP1.

These tools are contributed to STP, JWT and Scarbo. Note that these tools can be used whatever the service (webservices, RMI or SCA) runtime technology used (ex. Tuscany for the latest).

Section 6.1 describes the global trading and composing service architecture. Section 6.2 describes the abstract and concrete services specifications. Then, Section 6.3 presents the class in charge of the transformation of the abstract composition description, and finally, Section 6.4 specifies its different integrations in the SCOrWare project development tools.

6.1 The semantic system

Semantic composition of services using the Trading Service

6.1.1 Objectives

In the near future, services will be present everywhere and can be used by anyone. The diversity of services will allow a user to select the service -among many of the available services- which matches best his requirements and which is adapted most to the resources available to him. Service providers will provide services for a variety of devices having different capabilities. For an end-user, the choice of the best available service will not be so obvious. It is possible that a certain desired service is available to the user but he is not able to use it because it depends on some other services which are not available at the moment. The missing services can be replaced by similar available services, provided by a competitor; however a novice user does not know how to do that. In fact, the application has already been defined in terms of static services, so even if a replacement service is found, it cannot be used. On the other hand, if the user application is capable of replacing the missing service by another one dynamically, user need not to even aware of presence or absence of any services.

Such an application needs to be described in terms of abstract services which will be resolved into concrete services at the time of execution of the application, i.e. the service descriptions should be independent of their implementations.

6.1.2 Our Approach

The motivation behind our work is to propose a model for creating such applications whose composition, in terms of services, is static only at an abstract level and the exact concrete composition is determined at runtime. In this regard, we propose an infrastructure for dynamic composition of applications based on abstract service descriptions.

We would also like to mention that our contribution is not about proposing dynamic deployment of services nor do we plan to provide anything related to dynamic composition or orchestration of services. We assume that the application has already been defined in terms of the SCA composites, i.e. a composition of abstract services; however, we need to ensure the composition of application in terms of such concrete (SCA) services that best match the semantics of the abstract services. We also assume that the concrete services are already available during service selection process.

6.1.3 Trading and Composer Services Architecture

Trading is a technique, which consists in giving to a service component the possibility of dynamically discovering the components adapted to its needs (references). By Composing we meant the action of transforming the description of a composite, by replacing abstract subcomponents with concrete ones, so that the composite description, from abstract, become shallow or deep concrete. Six entities intervene in these processes: the service provider, the service consumer, the administrator and the the Registry , the SemanticTrader and the Composer (Figure 39).

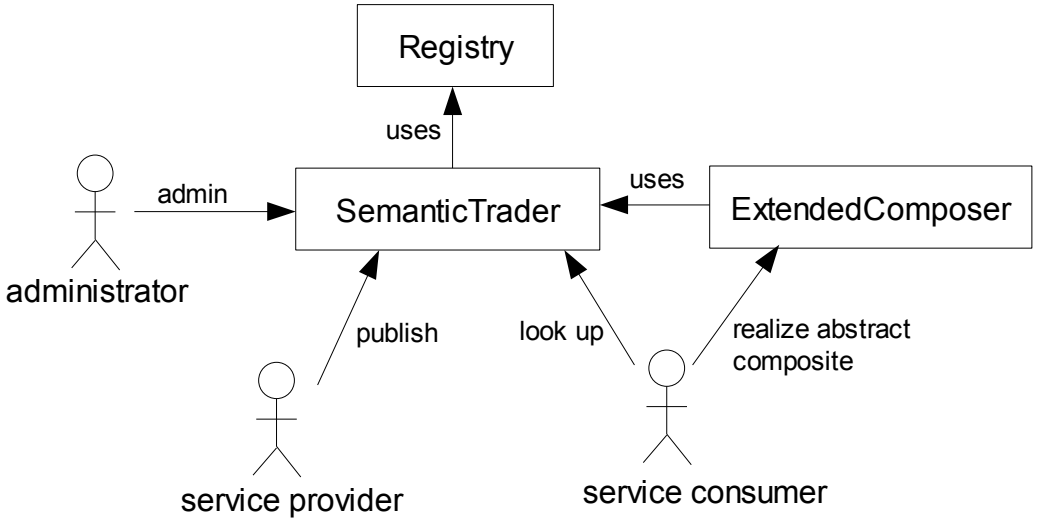


Figure 39: Trading Service

The semantic trader and composer play the role of intermediary between the other entities and thus allows them to interact in a distributed environment. The provider is the entity which uses the trader to publish information describing the services that it offers. The consumer is the entity which calls upon the trader to search services in the registry, or calls upon the composer to turn a composite from abstract to concrete. For the trader, even an incomplete description of the required component can be used in search. The registry is the entity used to store information describing the offered services, the comsumers and the providers. The administrator manages mainly the rights of access to the trading service (publishing and searching rights).

6.2 Abstract and Concrete Services Specifications

- Abstract Composite Description

In SCA, composites encapsulate services. A service offered by a composite is denoted by the service element in the composite. A required service is denoted by the reference element. To be able to reason about the services, we have to annotate services with reference to a semantic model, such as OWL or UML. This choice is motivated by the fact that applications developers can use any ontology language to annotate services (such as UML or OWL) unlike in OWL-S or WSMO. The description proposed in [12] is also available. The application domain is descibed as an algebraic data type : a set of sortes / types (with there subtyping relations), a set of constantes and operators, a set of equations that represents the objects propertie for this given application domain. Services are then represented as terms on this algebra. The match algorithm is based on E-unification.

Using SASCDL (see semantic trading service description, task 1.5), we suggest how to add semantic annotations to various parts of a SCDL document like services and their properties, components and their properties, and references. SASCDL defines a namespace called *sawsdl* and an extension attribute called *modelReference* so that relationships between SCA components and concepts in another semantic model are handled.

Figure 40 below shows the description of an abstract SCA composite. It only contains the service and component description as well as their properties. It also shows the semantic annotations added to each element.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"
   xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
   name="bigbank.account">

   <service name="AccountService"
      sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#Account">
      <reference>AccountServiceComponent</reference>
   </service>

   <property name="currency" type="xsd:string"
      sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#Currency">
   USD
   </property>

   <component name="AccountServiceComponent"
      sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#Account”>

      <property name="currency" source="$currency"
         sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#Currency"/>

      <reference name="accountDataService"
         sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountData">
      AccountLoggerDataServiceComponent
      </reference>
      <reference name="stockQuoteService"
         sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#StockQuote">
      StockQuoteServiceComponent
      </reference>
   </component>

   <component name="AccountLoggerDataServiceComponent"
      sawsdl:modelReference="http://www.int-
edu.eu/sawsdl/ontology/bigbank#AccountLoggerData">

      <reference name="accountDataService"
         sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountData">
      AccountDataServiceComponent
      </reference>
      <reference name="accountLoggerService"
         sawsdl:modelReference="http://www.int-
edu.eu/sawsdl/ontology/bigbank#AccountLogger">
      AccountLoggerServiceComponent
      </reference>
   </component>

   <component name="AccountDataServiceComponent"
      sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountData"/>
   <component name="AccountLoggerServiceComponent"
      sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountLogger"/>
   <component name="StockQuoteServiceComponent"
      sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#StockQuote"/>
</composite>
```

Figure 40: Abstract composite description

- Concrete Composite Description

The abstract composite description only specifies the services provided or required by it and does not show any detail on how the service is implemented, how to invoke it and which protocol should be used for this purpose. This essential information is provided in the concrete composite descriptions. A concrete composite description has the same set of services as defined in the abstract composite description and, in addition, also specifies the implementation details such as interfaces, classes, bindings and endpoints. Figure 2 shows the concrete composite description for the composite in figure 41.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"
   xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
   name="bigbank.account">
   <service name="AccountService"
      sawsdl:modelReference="http://www.intedu.eu/sawsdl/ontology/bigbank#Account">
      <interface.wsdl
         interface="http://www.bigbank.com/account#wsdl.interface(AccountService)"
         wsdli:wsdlLocation="http://www.bigbank.com/account wsdl/AccountService.wsdl" />
      <binding.ws
    endpoint="http://www.bigbank.com/account#wsdl.endpoint(AccountService/AccountServiceSOAP)"
         conformanceURIs="http://ws-i.org/profiles/basic/1.1"
         location="wsdl/AccountService.wsdl"/>
      <reference>AccountServiceComponent
       </reference>
   </service>
   <property name="currency" type="xsd:string"
       sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#Currency">USD
    </property>
   <component name="AccountServiceComponent"
      sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#Account”>
      <implementation.java
         class="bigbank.account.services.account.AccountServiceImpl" />
      <property name="currency" source="$currency"
         sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#Currency"/>
      <reference name="accountDataService"
         sawsdl:modelReference="http://www.intedu.eu/sawsdl/ontology/bigbank#AccountData">
      AccountLoggerDataServiceComponent
      </reference>
      <reference name="stockQuoteService"
         sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#StockQuote">
       StockQuoteServiceComponent
       </reference>
   </component>
   <component name="AccountLoggerDataServiceComponent"
       sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountLoggerData">
       <implementation.java
           class="bigbank.account.services.accountlogger.AccountLoggerDataServiceImpl"/>
       <reference name="accountDataService"
           sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountData">
       AccountDataServiceComponent
       </reference>
       <reference name="accountLoggerService"
           sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountLogger">
       AccountLoggerServiceComponent
       </reference>
   </component>
   <component name="AccountDataServiceComponent"
       sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountData">
       <implementation.java
           class="bigbank.account.services.accountdata.AccountDataServiceDASImpl"/>
   </component>
   <component name="AccountLoggerServiceComponent"
       sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#AccountLogger">
       <implementation.java
           class="bigbank.account.services.accountlogger.AccountLoggerServiceImpl"/>
   </component>
   <component name="StockQuoteServiceComponent"
       sawsdl:modelReference="http://www.int-edu.eu/sawsdl/ontology/bigbank#StockQuote">
       <implementation.java
           class="bigbank.account.services.stockquote.StockQuoteServiceImpl"/>
   </component>
</composite>
```

Figure 41: Concrete composite description

6.3 The ExtendedComposer

6.3.1 The ExtendedComposer architecture

The ExtendedComposer is the class in charge of the transformation of the abstract composition description. It is a "Facade" for the ShallowComposer and the DeepComposer which are respectively in charge of building shallow and deep concrete composites from abstract ones. They both rely on the SemanticTrader to retrieve concrete components before including them in the abstract composite in a way that can be parameterized (Figure 42).

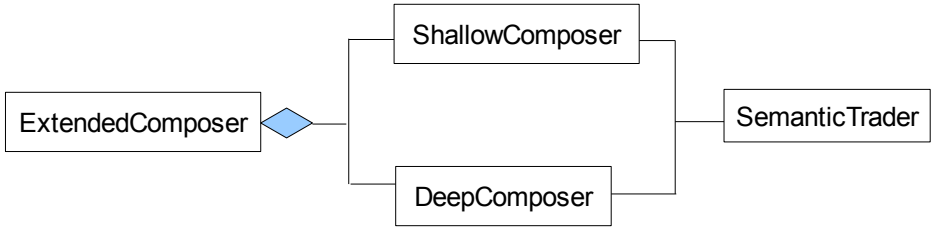


Figure 42: The ExtenderComposer architecture

6.3.2 The *ExtendedComposer* interface specification

The ExtendedComposer provides two main kinds of transformation of composite:

– the shallow transformation that makes composites shallow concrete by changing only direct children abstract subcomponents for concrete ones : through the **shallowAbstractToConcrete** methods,

– the deep transformation that makes composites deep concrete by changing all abstract subcomponents until leaves for concrete ones : through the **deepAbstractToConcrete** methods,

The ExtendedComposer is able to work either on composite files and in memory composites. The ModificationHistory object summarizes all the modifications made to a composite.

Composite **shallowAbstractToConcrete** (Composite abstractComposite)

This method builds a composite description, shallow concrete, semantically equivalent to the abstractComposite argument, by modifying its direct children. It takes as arguments the description instance of the composite that is shallow abstract (*abstractComposite*) and returns a composite description, semantically equivalent to the abstractComposite argument.

void **shallowAbstractToConcrete** (String abstractFile, String concreteFile)

This methods reads a composite description file, builds the description, makes it shallow concrete, and writes it to another file. It takes as arguments the original composite short file name (*abstractFile*) and returns the file in which write the result of the transformation (*concreteFile*).

ModificationHistory **getShallowModificationHistory** ()

This method returns the ModificationHistory of the last shallow transformation.

void **deepAbstractToConcrete**(String rootCompositeFileName, String sourceDir, String destinationDir)

This method rewrites a composite description and all it associated files in order to make the composite deep concrete. It takes as arguments the short file name of the composite description file (*rootCompositeFileName*), the directory containing files (for ex. : composite implementations of subcomponents, or "includes"...) referenced by the composite description (*sourceDir*), the directory in

which composite files are written (*destinationDir*).

void **deepAbstractToConcrete** (Composite rootComposite, String sourceDirectory)

This method modifies a composite description object in order to make it deep concrete. It takes as arguments the composite description object (*rootComposite*), and the directory containing the files (for example : composite implementations of subcomponents, or "includes"...) referenced by the composite description (*sourceDirectory*).

ModificationHistory **getDeepModificationHistory** ()

This method returns the ModificationHistory of the last deep transformation.

6.4 Tools for semantically-aided service composition design

6.4.1 Objectives

The aim is to provide tools for composing services and helping doing so by semantic search.

This is done by integrating INT's semantic service registry and search in the SCA Editor and in the JWT Editor, and as a prerequisite by allowing to design service composition in the JWT Workflow Editor.

6.4.2 Specification

Extensible JWT Metamodel

It is crucial in order to support service semantic search features in JWT that the JWT metamodel allows to be extended beyond its core definition. Only such an extension allows semantic annotations such as those used by INT's Service Semantic Trader to be put on JWT model elements (ex. A service-calling Application) without imposing it in all other use cases of JWT that don't use semantics.

In order to answer all similar requirements, and especially others within SCOrWare (like storing a transformed model's format-specific data, configuring runtime process or service engine-specific features), Open Wide has done a comprehensive use case study (http://wiki.eclipse.org/JWT_Metamodel_Extension_Specifications). It has concluded that in order to support features that can be transversal (related to any kind of model element) and optional, type inheritance is not enough and the following aspect-like architecture and developments are required.

A first version of model element-decorating aspects has been developed by Open Wide. It was designed such as :

- a JWT metamodel has a configuration (of its activated extensions)
- That has profiles (feature set)
- Each one specifying its Aspects (typed information, allowed to be instantiated in one or more model element types)
- That can be created in model elements extending ExtensibleElement (utility model element)

A second version has been refactored to be generic so it can work with any EMF model, and architectured as several Eclipse plugins. This is based on feedback (use cases, JWT partners...). Now a configuration (ConfModel) and its Aspect instances are stored in another file (*_conf), meaning that Aspects decorate and refer to model elements rather than exist within them. The (rather simple) metamodel is as follows (Figure 43).

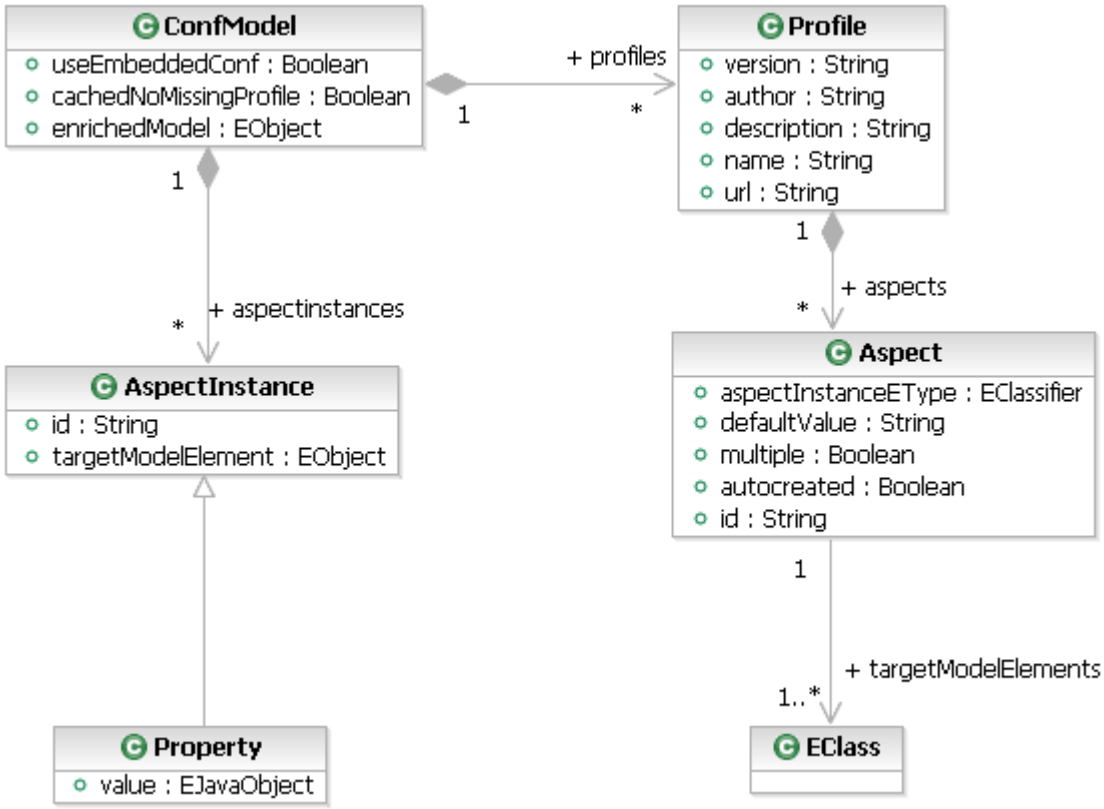


Figure 43: Extensible JWT metamodel part

JWT application model refactoring and service applications model design

In order to support service semantic search features in JWT, the JWT application model must be refactored and notably introduce a new ServiceApplication type, that will differenciate them from other kind of applications (such as Manual, Event-based...). This way, such ServiceApplications can be annotated by aspect-like extension (such as a SemanticAspect) to provide features specific to service-calling applications.

In order to make JWT fully benefit of SCA service technology agnosticity, the new model will not only comprise an ScaServiceApplication type, but also WebServiceApplication and RMIServiceApplication types.

JWT service applications GUI

In order to ease the basic design of ScaServiceApplication, WebServiceApplication and RMIServiceApplication in workflows, Open Wide improves their EMF-generated property sheet UI with various helpers.

SCA / SCOrWare service semantic provider

This is the service developed by the INT in lot 1.

SCA / SCOrWare service registry implementation and tools

The SCA / SCOrWare service semantic provider features semantic search algorithm but requires to be implemented on top of a service registry implementation. The aim of such a service registry implementation is to store and provide SCOrWare services, composites and their semantic annotations, i.e. Concretely SASCDL files (SCDL files along with the WSDL and SAWSDL files it needs).

This registry is useful

1. at development time to maintain consistency across the set of service used in an SCA architecture, by letting the development tools (like JWT or SCA Designer using the SCA component semantic search interface) peek in it
2. at runtime but only for the features that are using it, like the service semantic provider.

Note that implementation in the UDDI standard has been considered but rejected, because the SCA standard has service definitions that goes beyond what UDDI knows how to manage – and that's without even speaking of storing semantically annotated SCA composite definitions or of UDDI excessively complex and limited extension capabilities.

In addition, Open Wide develops a simple tool to graphically add a given composite to the registry.

Ontology GUI

An Eclipse view allowing to search and select semantic concepts has been developed by EBM WebSourcing using the Jena semantic library, contributed to Eclipse STP in a generic manner with the help of an additional Jena-dependent plugin that is contributed to OW2.

In a first version, its underlaying ontologies (.owl files) are configured on the same view.

In its second version, this configuration UI is shared in the form a single preferences interface with the help of Open Wide among other SCOrWare semantic tools (see below).

Semantic annotation of services in the STP SCA Editor

Semantically annotating SCA services is possible through the drag-and-drop of an ontology concept from the ontology view on a service displayed in the SCA Editor. This integration is developed by Obeo.

Semantic annotation of service applications in the JWT Workflow Editor

Semantically annotating SCA services is possible through the drag-and-drop of an ontology concept from the ontology view on a service application displayed in the JWT Workflow Editor.

This integration is developed by Open Wide and contributed to JWT, using Eclipse drag-and-drop. The semantic annotation is stored on a Service Application in a dedicated SemanticAnnotation EMF aspect. Moreover such drag-and-drop is made more flexible in JWT by defining a dedicated extension point.

SCA / SCOrWare service semantic provider integration as SCA component and tools

This is a way to easily use an SCA component wrapping the SCA / SCOrWare service service semantic provider and allowing to use it at runtime to call a development time semantically described, runtime chosen service. It has been developed by the INT in lot 1.

The STP SCA Designer by Obeo allows to use and configure said SCA component and the service semantic provider service it wraps, just like any SCA component.

Semantically annotated SCA service application resolution to composite in JWT at design time

A semantically annotated SCA service application represents the call of an “abstract” SCA service, that is of a service defined on an “abstract” SCA composite that is not known yet, and must be resolved before execution by the INT trading service using the provided annotation within the defined ontology context.

We specify that JWT allows to resolve it at design time, in order to help users start from template workflows and from there resolve their annotated target services within specific ontology contexts. In order to to this, Open Wide designs a JWT External Action that can be triggered on an SCAServiceApplication and that asks the Eclipse-integrated INT Trader to resolve its target concrete SCA composite, by using a composite generated from the description of the SCAServiceApplication and containing the annotation found in its SemanticAnnotation aspect

Semantically annotated SCA service application resolution to composite in JWT at runtime

Resolution at runtime is possible thanks to the INT Trader being integrated to FraSCAti, at the condition it is provided with a semantically annotated composite. This is allowed in JWT by designing an SCA Service Application whose "compositeFile" attribute refers to a semantically annotated composite definition. This requires the chosen SCA runtime technology to be SAWSDL aware, like FraSCAti is.

It is worth mentioning that this could also be allowed on non-SCA service applications (like WS and RMI), by letting the Scarbo implementation of the TaskEngineFramework call FraSCAti on composites that are generated using the library and aspect mentioned above. This requires to use the Scarbo runtime and its FraSCAti-based SCA integration.

Integration of service semantic search interface in the SCA Designer

See in the SCA Designer part.

Semantically annotated composite Editor

This editor completes the registry feature set.

The STP SCA Editor has been extended, notably by Obeo, in order to support semantic annotations.

Integration of scientific computing dedicated semantic service trader

Two kind of solution can be used in the semantic trading of services for deciding if a given service fits the requirements of the user: general solutions relying on existing WEB SEMANTIQUE technologies which can be used in any context but are limited in their expressiveness; and domain dedicated solutions which can use the specific semantic properties of the application domain in order to have sophisticated description of the services and comparison procedures.

IRIT will adapt in SCOrWare the scientific computing dedicated semantic service trader that have been developed in the TLSE project. IRIT will rely on SCOrWare semantic description formalism in order to define the properties required for its trading technology and will adapt its trading algorithm to use this new description and provide the results which will be used in the graphical wizards.

6.4.3 Implementation

Extensible JWT Metamodel

It is developed by Open Wide so as to provide many useful features, such as :

- Library allowing to know and manage installed Profiles, but also a given model's Profiles, and its element's Aspects instances
- Conf (Profiles and Aspects) and Aspect instances are stored in a separate _conf file along the EMF model they decorate. Renaming and moving them together is supported.
- Configuration through extension point or autodiscovery directory
- Aspects can be designed to be automatically created when a model element is created, be singleton or multiple, have a default value
- Works with dynamic as well as statically generated EMF models
- Extended information (Aspect instances) can be created and shown on the model element it decorates, thanks to EMF.Edit customization
- Profile Management GUI, as a dialog or editor tab
- Development mode ("embedded mode") and various aspect design tools, more are planned (outside of SCOrWare)
- Provided with a sample simple, all-around key-value "Property" model and its property sheet UI
- In a separate plugin, JWT-specific developments integrate it in the Workflow Editor

Details of implementation are available at https://bugs.eclipse.org/bugs/show_bug.cgi?id=241567 . Documentation

is available at http://wiki.eclipse.org/JWT_Metamodel_Extension .

This is contributed to JWT. Note that it can be useful to any EMF model that needs to be decorated by additional typed information.

JWT application model refactoring and service applications model design

Open Wide does this with the help of JWT partners using the EMF MDA toolchain, the JWT Conf metamodel extension framework and JWT's GEF-based editor. SCAServiceApplication, WebServiceApplication and RMIServiceApplication are designed as Aspects within an installed Conf extension.

JWT service applications GUI

In addition to the default EMF generated UI to model service applications, various helpers are developed by Open Wide, as follows.

- A dialog box allows to choose an SCA service in an SCA composite file, thanks to SCA introspection based on the STP SCA EMF model.
- Double-clicking on an SCAServiceApplication opens the STP SCA Editor. In order to develop this in a generic manner, an extension point allowing to hook double-click behaviour on a given JWT model element type is defined.
- Another allows WSDL retrieval and introspection to let the user choose the service operation he wants to call, but also to generate service parameter mappings (with workflow variables) thanks to the OW2 EasyWSDL library. Note that EasyWSDL IP issues prevents it to be contributed to Eclipse JWT, but it is at least properly decoupled from JWT through JWT's generic PropertyDescriptor extension point.

All previous service application modeling tool are contributed to JWT, save for WSDL tooling which is contributed to Scarbo. Note that these tools can be used whatever the service (webservices, RMI or SCA) runtime technology used (ex. Tuscany for the latest).

SCA / SCOrWare service registry implementation and tools

Such a registry implementation, as well as administration tools, has been developed by INT for its semantic service Trading in Java on top of a configurable persistence layer, that can use a database, or merely a flat file or reside in memory.

Open Wide integrates it as a Scarbo-contributed Eclipse plugin.

Open Wide develops a simple action within Eclipse to graphically add a given composite to the registry, and contributes it to Scarbo.

Ontology GUI

An Eclipse view allowing to search and select semantic concepts has been developed by EBM WebSourcing using the Jena semantic library, contributed to Eclipse STP in a generic manner with the help of an additional Jena-dependent plugin that is contributed to OW2.

In a first version, its underlaying ontologies (.owl files) are configured on the same view.

In its second version, this configuration UI is shared in the form a single preferences interface with the help of Open Wide among other SCOrWare semantic tools (see below).

Semantic annotation of services in the STP SCA Editor

Semantically annotating SCA services is possible through the drag-and-drop of an ontology concept from the ontology view on a service displayed in the SCA Editor. This integration is developed by Obeo and contributed to STP.

Semantic annotation of service applications in the JWT Workflow Editor

Semantically annotating SCA services is possible through the drag-and-drop of an ontology concept from the ontology view on a service application displayed in the JWT Workflow Editor.

This integration is developed by Open Wide and contributed to JWT, using Eclipse drag-and-drop. The semantic annotation is stored on a Service Application in a dedicated SemanticAnnotation EMF aspect. Moreover such drag-and-drop is made more flexible in JWT by defining a dedicated extension point.

SCA / SCOrWare service semantic provider integration as SCA component and tools

This is a way to easily use an SCA component wrapping the SCA / SCOrWare service service semantic provider and allowing to use it at runtime to call a development time semantically described, runtime chosen service. It has been developed by the INT in lot 1.

The STP SCA Designer by Obeo allows to use and configure said SCA component and the service semantic provider service it wraps, just like any SCA component.

Semantically annotated SCA service application resolution to composite in JWT at design time

This is implemented as an Eclipse Wizard. Helper design time tool include opening the service application resolution wizard on double-click (by hooking on the JWT model double-click extension point) on an annotated service application. Annotated composite generation is achieved using Open Wide's SCA composite generation library.

Open Wide has integrated INT's Trader as an Eclipse plugin, first in year one and then updated to the final version of the Trader, to be provided in the Scarbo project. In year one, Open Wide has developed a simple SCA service semantic search UI to help prototyping use cases. In year two, Open Wide has implemented the above specifications and contributed them to Scarbo.

The previous semantic service tools are contributed to STP, JWT and Scarbo. Note that these tools can be used whatever the service (webservices, RMI or SCA) runtime technology used (ex. Tuscany for the latest).

7 Choreography of process that use generic services

The main objective of the task 2.6 is a workflow and orchestration solution (tooling in Eclipse JWT and runtime in Scarbo and its default Bonita workflow engine) that is able to work with “generic” services by taking advantage of SCA / SCOrWare service technology agnosticity, built on the SCOrWare-powered JWT – Scarbo toolchain, and flexible enough to meet development time and runtime requirements (as defined in demonstrators).

Flexible choreography is achieved at runtime by the semantic process matcher and its registry. Design time tooling for flexible choreography has also been studied, as well as decoupled runtime process registry APIs that ease its integration with JWT.

7.1 “Generic” SCA / SCOrWare service integration in BP solution

7.1.1 Objectives

The aim is to provide BP tooling for “generic” services, that is services beyond “mere” web services, thanks to the cross platform and cross language capacities of SCA and especially SCA service bindings.

This generic service tooling in processes is achieved through SCA service integration within JWT at design time, and within Scarbo's TaskEngineFramework and its default Bonita process engine at runtime, in a consistent manner.

7.1.2 Specifications

Preliminary Note

This task has been affected by Amadeus' departure from the project. As a consequence, the focus has been less on requirements provided by the Amadeus demonstrator, including classic entreprise integration connectors as SCA bindings (JMS, EDI...), and more on fully taking advantage of FraSCAti technology in JWT, as well as requirements of the new Thalès demonstrator. However, on the binding side, note that Scarbo can execute at runtime SCA, RMI and WebService bindings that have been defined in JWT, and beyond this the FraSCAti-integrated PetALS ESB itself provides a host of connectors.

Service tools in JWT process definition and modeling (see previous chapter)

Tools allowing in the JWT Workflow Editor to model (thanks to a metamodel extension) and define (with the help of specific tools like dialogs and WSDL introspection) Applications calling webservices, RMI services and SCA-defined services have been specified in the previous chapter and developed by Open Wide with the help of the JWT community. These tools are contributed to JWT

Additional tools helping design such service calls within processes, like semantic service annotation and resolution tools, as well as their configuration UI, have also been specified in the previous chapter and developed by Open Wide, EBM WebSourcing and Obeo. These tools are contributed to JWT, STP and Scarbo.

Note that these tools can be used independently of the service (webservices, RMI or SCA) runtime technology (ex. Tuscany for the latest).

SCA / SCOrWare service integration in JWT process execution engine

This will be done by Open Wide in the context of the SCOrWare project, with the help of the JWT community.

JWT processes will be able to call SCA services.

Conversely SCA services will be able to call JWT processes, and react on JWT events.

In year one, we will simply develop support for calling SCA services within JWT process execution.

JWT Runtime APIs – WorkflowService and the TaskEngineFramework

JWT Runtime APIs are obviously required to provide runtime tooling in JWT (such as process monitoring GUI). Runtime tooling in JWT needs to manage workflows and especially states of their instances within a process engine, which requires JWT to define a common API to be implemented on top of target process engines. Such an API is embodied at a simple level by the JWT Runtime WorkflowService. Note that more useful APIs on this side should cover administration tasks and notification.

However that's far from all. Let us remind that JWT's focus is laid on providing flexible business process tooling independently of standards and runtime. Beyond JWT"s flexible architecture, we've seen that modeling flexibility is provided by its extensible metamodel (through EMF inheritance and the Conf framework aspect-like decorations) which allows any kind of features to be modeled (in the context of SCOrWare, such as service, SCA or semantics oriented features), while support of standards and executable processes is provided by JWT Transformation plugins (like JWT to XPDL transformation). But to fulfill JWT's promise, it has to be defined how custom features designed using tool extensions and modeled thanks to the extensible metamodel should be executed at runtime on any process engine.

From there comes the idea of providing a simple framework and APIs that are made easy to integrate on top of any java process engine, and outside of this scope still demonstrate how providing runtime support to JWT extended features should be achieved. Such APIs that decouple workflow task implementation from process engines include the ProcessModelService, that gives access to task (action) implementation configuration (including metamodel extensions, in the form of simple pathed properties, besides the specific case of parameter mapping) as it was designed in the workflow definition, and ProcessStateService, that allows to manage an instanciated workflow's current state (as a simple key-value map).

We've even gone further in SCOrWare. Indeed, we use the FraSCAti service platform to provide the ability to execute a workflow's SCAServiceApplication, but also RMIServiceApplication and WebServiceApplication. Said another way, we allow to configure existing FraSCAti bindings transparently in JWT. From there comes the idea in the specific field of SOA-aware processes to decouple service tasks (actions) from the service platform implementation. This is done by the ServiceProvider API.

These APIs, along with helper code including sampled default implementations, make up the JWT TaskEngineFramework.

Architecture of the TaskEngineFramework and of Scarbo's implementation

The OW2 Scarbo project has been specified to implement JWT's generic TaskEngineFramework on top of SCOrWare-developed and OW2-hosted middleware components : notably FraSCAti and Bonita.

The whole JWT TaskEngineFramework and Scarbo architecture is as in Figure 44. APIs are in yellow, and the picture includes their Scarbo specific implementations on top of Bonita and FraSCAti.

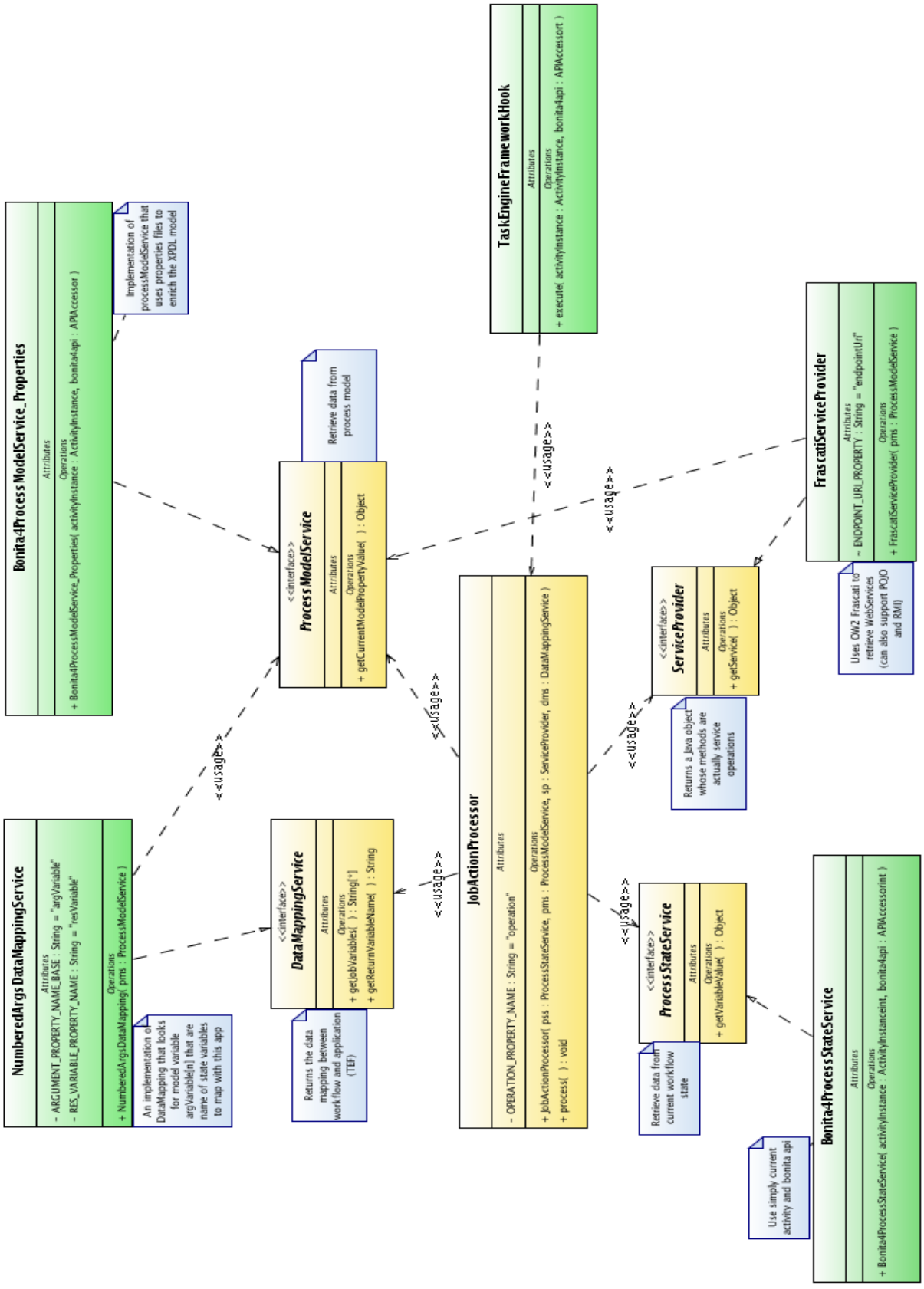


Figure 44: JWT TaskEngineFramework and Scarbo architecture

Data mapping in JWT for SOA / Scarbo

Data mapping is useful in JWT for SOA / Scarbo to design how to map service call complex parameters from parts of complex process variables, and letting this mapping be done at runtime when actually calling the service. Mapping configuration consists typically of a collection of source-to-target mappings, with source and target being expressions to allow complex type introspection. However, mapping can also be done by any other kind of technology, be it specific to a given Action (task) implementation, or openly coded in any kind of language or script.

In JWT for SOA, we logically focus on Java runtimes. Therefore, powerful mapping is provided easily at runtime by integrating a Java-embeddable scripting engine. It allows to script at once any kind of mapping between variables of a process. Note that it can be further used to resolve defined source and target parameter mappings.

Therefore Open Wide integrates the Groovy scripting engine in Scarbo's TaskEngineFramework, which is the perfect place for it in an architectural point of view, defines a JWT metamodel extension consisting in Aspects allowing to define scripts defined as Applications and on Actions (tasks) as well as their language (from which a target engine can be chosen at runtime) and target data domain (allows to maintain consistency between the types of engine-managed state variables and their use in Applications, scripts... By default the Java data domain, for this JWT DataTypes can be used).

About Data binding in JWT for SOA / Scarbo

SCA drives a line between “protocols” and “data binding”. We note that data binding is first and foremost and integration issue, and that the choices of SCOrWare and also more globally of the markets of SCA, ESB, JBI and service-oriented integration are targeted toward XML-compatible data binding.

For the record, common java and XML databinding encompass standards like bare or typed (through schemas like XSD, RelaxNG) XML as well as DOM, SDO, JAXB, are connected to various XML transformation, marshaling and mapping technologies, and implemented by components like the JDK, CXF, or even standalone technologies like Apache XMLBeans.

In the JWT for SOA tooling and Scarbo solution, we make the (logical) choice to focus on Java runtimes. This allows to take advantage of FraSCAti as a native Java implementation of the SCA standard, which is service technology-agnostic. That means that by providing integration on the Java level between FraSCAti and java process engines, which has no databinding issues in itself, FraSCAti provides a databinding solution to any supported service technology. For instance, when we want to call a WebService, we design an SCA WebService binding, provide its required call parameters as Java beans, and let FraSCAti map them to the right typed XML.

Note that EBM WebSourcing has implemented JBI and XML to FraSCAti databinding for its FraSCAti integration within its PetALS ESB.

About Events in JWT for SOA / Scarbo

In an SOA world, emission of an event is seen as the call of services that listen to it.

The emission of a service-oriented event in a process can be designed in JWT by defining one or more Action(s) that is (are) executed by a ServiceApplication that configures the call .

Open Wide defines and provides the WSDL definition and Java artifacts of a technical, all-purpose EventListenerService to simplify the use of events in the case where the user has not already defined an appropriate business-level event listening service.

Note that extended event support would include two things that are here out of scope. First, the kind of event (for workflows which are conditioned transitions linking black-box tasks, these are exclusively task events : onReady for manual activities, before/afterStart, before/afterTerminate) should be able to be defined ; however the various kind of events can be simulated by designing an additional service calling Action before or after the Action to hook the event on. Second, to make fully configurable what event a listener receives, it should be possible to provide a dynamic boolean condition (most powerfully defined by code : typically a scripted expression) whose evaluation triggers or not the emission of this event ; for triggering such service-oriented, conditioned event, Scarbo's TaskEngineFramework's JobActionProcessor implementation would be the perfect place.

7.1.3 Implementation

Introducing the OW2 Scarbo project

Scarbo is an open, SOA ready, SCA powered BPM solution built on OW2 and Eclipse.

At its origins, developments tied to FraSCAti and Bonita APIs are LGPL'd and can't go in Eclipse out of license incompatibility, but more crucially and on a higher level, JWT tooling wants to stay technology agnostic, and therefore Scarbo is “only” one possible implementation of JWT for SOA features, albeit one that uses SCOrWare-produced and OW2-hosted technology.

So : Scarbo brings technology agnostic Business Process design and execution to your service architecture. Built on Eclipse Java Workflow Tooling (JWT) and the FraSCAti SCA platform, it supports the Nova Bonita workflow engine while being open to others.

What does Scarbo consist in ? It builts on top of the Eclipse Java Workflow Tooling (JWT) generic tooling suite, the OW2 Bonita workflow engine and the SCA-implementing FraSCAti service platform, as well as other tools developed and demonstrators by fellow partners of the SCOrWare ANR project. It is actually the implementation of JWT runtime APIs and more widely of the "JWT for SOA" vision that uses the SCOrWare service platform as its runtime, and therefore a good JWT integration showcase as well.

Providing JWT for SCOrWare runtime features : Scarbo's Task Engine Framework implementation

It consists of :

- an implementation of the JWT WAM Task Engine Framework's Engine API (ProcessStateService, and a JobActionProcessor that can be executed as a Bonita Hook) on top of the OW2 Nova Bonita workflow engine.
- a default implementation of the ProcessModelService that relies on a simple property file to store the process definition configuration properties. This is a simple, powerful solution that bypasses any limitations at the process engine level in storing any kind of process definition extended properties, without preventing custom, more efficient implementations that sit directly atop process engines. This property file must have been generated, typically during a JWT Transformation, and helper tools to do exactly that are also provided.
- an implementation of the JWT Runtime Task Engine Framework's Service API (ServiceProvider) on top of OW2 FraSCAti. Beyond this implementation, there are many ways Scarbo can work with other service platforms, be it through simple web services, SCA services, or even JWT WAM TEF implementation.
- This FraSCAti-based implementation of ServiceProvider relies on a portable (thanks to its own classloader) integration of the OW2 FraSCAti open SCA implementation in the OW2 Nova Bonita workflow engine, flexible SCA composite generation, and Java webservice artifact generation (), which are all provided as libraries on their own.
- NB. also provided are additional (notably SCOrWare-specified) SOA and BPM oriented features, tools (ex. SCA integration, semantic service search, a workflow monitoring Eclipse plugin) and demonstrators building on those.

This allows to execute processes that call SCA service, webservices and RMI services as designed using JWT.

SCA composite generator

JWT for SOA / Scarbo provides it as a library to allow to easily generate simple SCA composites out of design-time configuration.

It is used in the Scarbo runtime in order to configure the call by FraSCAti of design-time configured WebServices or RMI services, by generating a composite containing a single service definition, proxy implementation and external reference binding.

It is used in the design-time resolution of semantically annotated abstract services applications against INT's Trader,

by generating an abstract composite containing a service annotated by the semantic concept defined in the decorating SemanticAspect.

Java webservice artifact generation

Scarbo provides a library that generates the required Java webservice artifacts using the Java to WSDL algorithm provided by CXF, which is the WebService engine used by FraSCAti to support WebServices).

Note that such generation is common and useful outside the scope of SCOrWare. Other SCOrWare partners like EBMWebSourcing have worked on similar generation features, be it at runtime (including the issue of going across the various standards of the databinding field like POJO and JAXB) in various engines or at design time in Eclipse or even command-line tools.

That's why there are ongoing efforts by these actors to unify them in a common approach in Eclipse STP, which is the right place for such service-oriented tooling.

Therefore Open Wide integrates the Groovy scripting engine in Scarbo's TaskEngineFramework, which is the perfect place for it in an architectural point of view, and defines a JWT metamodel extension consisting in Aspects allowing to execute scripts defined in Actions (tasks).

WorkflowService for process engine monitoring in JWT for SOA / Scarbo

JWT Runtime APIs' WorkflowService is useful to let external business applications interact with ongoing workflows, and especially “manual” applications where an operator chooses tasks to do from a to-do list and manually does them. This has been showcased in the OS2P demonstrator.

Open Wide has developed a simpler demonstration of it that is useful for everyone and consists in an Eclipse GUI (View) allowing to see the current state of a target process engine, and provided it as a JWT plugin.

Data mapping in JWT for SOA / Scarbo

Open Wide integrates the Groovy scripting engine in Scarbo's TaskEngineFramework, which is the perfect place for it in an architectural point of view, and defines a JWT metamodel extension consisting in Aspects allowing to execute scripts defined in Actions (tasks).

About Events in JWT for SOA / Scarbo

Open Wide defines and provides the WSDL definition and Java artifacts of a technical, all-purpose EventListenerService to simplify the use of events in the case where the user has not already defined an appropriate business-level event listening service.

7.2 Process semantic matching

7.2.1 Objectives

The aim is to be able to match “abstract” (i.e. That has abstract steps, i.e. In BPEL with missing partner links) process definitions at runtime, so when such an abstract process is executed at runtime, its abstract steps are resolved to another, collaborative process that implements them, and this collaborative process can also be instanciated and used to complete the first one.

7.2.2 Specification

This process matching is done semantically on BPEL processes without partner links. Ordered Binary Diagrams (OBD) of processes are used instead of the processes themselves to improve matching efficiency. Processes and OBDs are stored in a runtime registry. Once the abstract steps of a given process “A” are resolved through semantic matching to be those of another abstract process “B”, both processes are rewritten and their partnerLinks completed

to be each other's services, so they can be executed successfully in a collaborative manner.

See more about it in INT's related published articles.

7.2.3 Implementation

This process matching has been implemented on a BPEL process engine by INT.

Note that it has also been implemented by INT outside of SCOrWare on various workflow engines (such as Shark).

See more about it in INT's related published articles.

7.3 Process semantic matching in JWT

7.3.1 Objectives

The aim will be to provide tools and study how to best and usefully integrate INT's semantic process matching and choreography engine within JWT.

7.3.2 Specification

Tools for designing abstract processes

Abstract processes are processes containing abstract steps, that is steps that have no implementation on their own but are rather meant to be implemented by a step of another, collaborative, matching process.

In order to be able to design such abstract processes in the JWT Workflow Editor, an abstract Action type has to be defined.

Note that Bull, a JWT partner, as worked to allow to design in JWT abstract tasks and afterwards add implementations details. It is not the same use case of an “abstract” step, but worth mentioning, and the possibility of a common approach of it will be studied (outside of ScOrWare).

Introduction

This will be done by Open Wide with the help of the INT. The aim is to help develop SCA composites using SCA / SCOrWare services, by

1. Allowing to graphically search SCA / SCOrWare services corresponding to a certain set of semantic criteria and
2. Allowing SCA composites to graphically integrate and configure the semantic service courtage service, so as to semantically define a composition of services to call.

It will be developed as an Eclipse plug-in to the JWT designer and will use the JWT / SCOrWare process registry and process semantic provider (matcher ?).

INT's semantic process registry and JWT at design time

INT's semantic process registry's aim is to store and provide JWT / SCOrWare processes and their semantic annotations.

Open Wide has studied its use to provide at design time tools that allow to fine processes in order to complete an abstract process with collaborative tasks copied from a matching process. Anyway, the JWT Workflow Editor follows the workflow (oriented graph) paradigm, whereas INT's process matching technology works with BPEL (block-based language) processes. So incompatible technologies make integrating both require a high cost and not being much useful. That's why such development is not released in the SCOrWare project.

Open Wide wide has validated the possibility of using the Eclipse Technology BPEL Editor subproject to design

appropriate abstract BPEL processes. This is even possible out-of-the-box, since abstract BPEL processes are merely BPEL processes with some missing partnerLinks.

Open Wide has also studied the possibility of decorating in JWT an “abstract” subprocess with information required to define an abstract BPEL process' OBD trace, so they could be resolved by matching to a concrete BPEL subprocess at design time or runtime. Such information would have to be a list of abstract steps. However no use case has been found that make this interesting enough to develop it.

INT's semantic process registry and JWT at runtime

Since Open Wide's and the INT's runtime use different process engines that execute processes defined in incompatible languages, there is little point in natively integrating them.

However it is possible to make both work together at runtime, by defining BPEL processes on one side and JWT workflows on another that call each other's WebServices, although process matching obviously can't resolve to JWT workflows.

Finally, Open Wide studies and specifies JWT Runtime-hosted APIs for an open process registry.

JWT / SCOrWare process semantic provider

This service is provided by the process semantic registry.

7.3.3 Implementation

Tools for designing abstract processes

Open Wide defines and implements as specified an extension to the JWT metamodel allowing to define an “abstract” JWT Action. It is done as an Eclipse plugin comprising an EMF Aspect and its EMF metamodel, using the EMF MDA toolchain, the Conf framework and the EMF – GEF editor.

INT's semantic process registry and JWT at runtime

Open Wide studies and specifies JWT Runtime-hosted APIs for an open process registry, that could allow INT's semantic process registry to be integrated in JWT, but also first and foremost fulfills typical process registry use cases like process definition deployment and management.

7.4 Model transformations

7.4.1 Objectives

This tool generates BPMN code from JWT code, and JWT code from BPMN code.

7.4.2 Specification

The JWT meta model is presented Figure 27 and the BPMN meta model in Section 3.2.2.

The following table depicts the mapping that we propose between the main concepts of BPMN and JWT. To define this mapping, we adopt the most relevant aspects for a comparison of Business Process Meta models proposed in [11]. This article presents a comparison between a simple Business Process Meta model, BPBM (Business Process Definition Meta model), EPC (Event-driven Process Chains), List/Korherr, UML2 Activity Diagram and JWT.

Concept	BPMN	JWT
General concepts		
Process	Business Process Diagram	Activity
Process behavior	?	No distinction
Link to another process	Collapsed Sub Process Expended Sub Process	Activity Link Node
Included process	Collapsed Sub Process Expended Sub Process	Structured Activity Node
Group	?	Group
Activity	Task Activity	Action
Transition	Flow ?	Activity Edge
Guard on transition	Event ?	Guard/Guard Specification
Loops	Activity Looping Sequence Flow Looping Multiple Instances	-
Control nodes		
Process start	Start Event Start Message Event Start Timer Event Start Rule Event Start Link Event Start Multiple Event	Initial Node
Process finish	End Event End Message Event End Error Event End Compensation Event End Link Event End Terminate Event End Multiple Event	Final Node
Process flow abort	Intermediate Cancel Event End Cancel Event	No distinction
XOR split	Data Based Gateway	Decision Node
XOR join	Data Based Gateway	Merge Node
AND split	Parallel Gateway	Fork Node
AND join	Parallel Gateway	Join Node
OR split	Inclusive Gateway	-
OR join	Inclusive Gateway	-
IOPE		
Input data	Event Message Flow	Data
Output data	Event Message Flow	Data
Events		
Event	Event	Event

Message event	Start Message Event Intermediate Message Event End Message Event	Message Event
Timer event	Start Timer Event Intermediate Timer Event	Timer Event
Rule event	Start Rule Event Intermediate Rule Event	-
Link event	Start Link Event Intermediate Link Event End Link Event	-
Multiple event	Start Multiple Event Intermediate Multiple Event End Multiple Event	-
Compensate event	Compensation Association Association Intermediate Compensation Event End Compensation Event	-
Error event	Intermediate Error Event End Error Event	-
Business specific		
References	-	Reference
Business function	-	Function
Role	Pool Lane	Role
Organization	-	Organization Unit
Application	-	Application
Parameter	-	Parameter
Interactions		
Interaction	?	-
Message channel	Event	-
Interaction role	Pool (when the pool is used as a black box)	-
Flow binding	Input Maps attribute of Sub Process Output Maps attribute of Sub Process	-

Figure 45: BPMN/JWT Mapping

7.4.3 Implementation

ATL is used to generate BPMN from JWT and JWT from BPMN according to the mapping defined in the previous section.

8 Conclusion

This document specifies tools developed by the different SCOrWare partners. Some of these tools do not aim a particular SCA runtime but complete existing tools of the STP project, the JWT project, or the Scarbo project:

- SCA Creation Tools (Section 3.1),
- SCA Editors (Section 3.2),
- Convenience Tools for Developers (Section 3.3),
- SCA Composite Designer (Section 4.1),
- JWT Business Designer (Section 4.3),
- JWT Technical Designer (Section 4.4),
- Tools for deployment (Section 5.1),
- Tools for test (Section 5.2).
- Tools for Searching and Semantic Composition of Services (Section 6),
- Transformation between JWT and BPMN (Section 7.3).

All these tools simplify the work of architects, analysts, and developers by automating tedious tasks at different steps of the SCA applications construction. Figure 46 shows an overview of different roles and SCOrWare tools involved in the development of SCA applications.

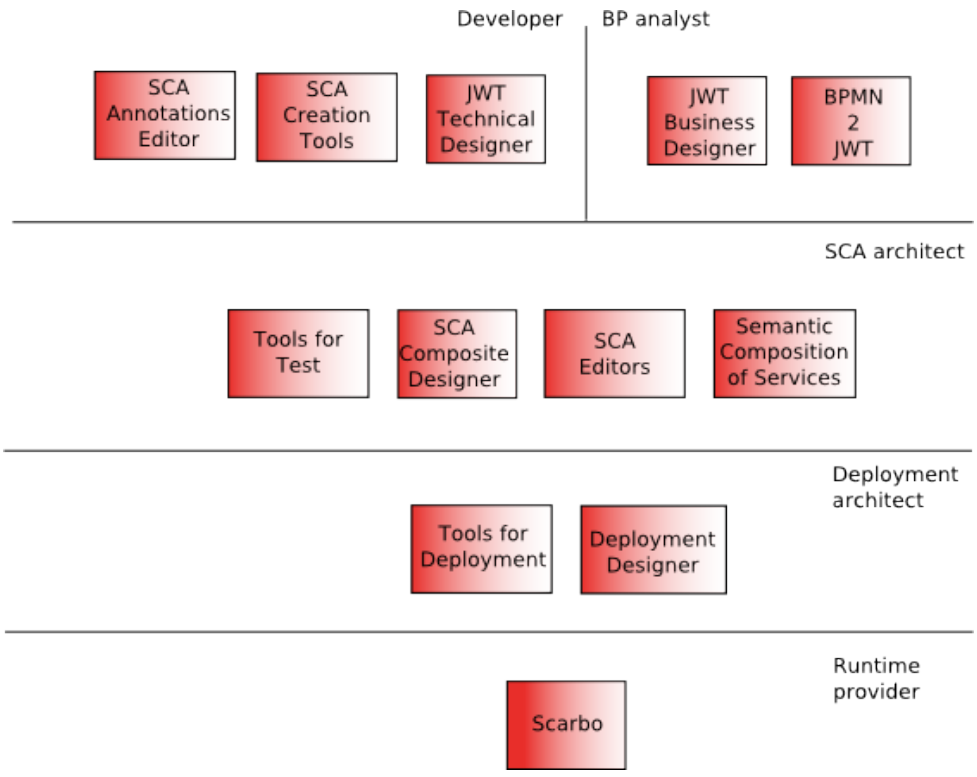


Figure 46: Roles and tools involved in SCA applications development

9 References

1: OMG, OMG Model Driven Architecture, http://www.omg.org/mda/

2: Wikipedia, Domain-specific programming language, http://en.wikipedia.org/wiki/Domain-specific_programming_language

3: The Eclipse Foundation - STP Project, SOA Tools Platform Project (STP), http://www.eclipse.org/stp/

4: Philippe Merle and all, SCOrWare Project - SCA Platform Specifications - Version 2.0, March 2009

5: The Eclipse Foundation - EMF Project, Eclipse Modeling Framework Project (EMF), http://www.eclipse.org/modeling/emf/

6: Open Service Oriented Architecture collaboration, SCA Assembly Model Specification V1.00, 2007

7: The Eclipse Foundation - EMF Project, The Eclipse Modeling Framework (EMF) Overview, http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html

8: The Eclipse Foundation - EMF Project, The EMF.Edit Framework Overview, http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.Edit.html

9: The Eclipse Foundation - EMF Validation Sub Project, Eclipse Modeling Development Tools subcomponent Validation, http://www.eclipse.org/modeling/emf/?project=validation#validation

10: The Eclipse Foundation - GMF Project, Eclipse Graphical Modeling Framework (GMF), http://www.eclipse.org/gmf/

11: Florian Lautenbacher, Comparison of Business Process Metamodels, 2007

12: Aurélie Hurault, Michel Daydé, Marc Pantel, Advanced service trading for scientific computing over the grid, The Journal of Supercomputing